intel®

Intel® Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 4: Cryptography

Document Number: 303881-05

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-01	Documents Intel® Integrated Performance Primitives (Intel® IPP) 4.0 beta release.	05/ 2003
-02	Documents Intel® Integrated Performance Primitives (Intel® IPP) 4.0 gold release.	10/2003
-03	Documents Intel® Integrated Performance Primitives (Intel® IPP) 5.0 release with ECC and DL-based functionalities.	03/2005
-04	Documents Intel® Integrated Performance Primitives (Intel® IPP) 5.0 gold release.	08/2005
-05	Documents Intel® Integrated Performance Primitives (Intel® IPP) 5.1 gold release.	02/2006

Contents

Bibliography

Index

Examples

Overview 1

This manual describes the structure, operation, and functions of Intel® Integrated Performance Primitives (Intel® IPP) for cryptography. This is the fourth volume of the Intel IPP Reference Manual, which also comprises descriptions of Intel IPP for signal processing (volume 1), Intel IPP for image and video processing (volume 2), and Intel IPP for small matrix operations (volume 3).

The Intel IPP software package supports many functions whose performance can be significantly enhanced on the Intel® Architecture (IA), particularly using the MMX™ technology and Streaming SIMD Extensions.

Intel IPP for cryptography is a cross-platform software layer optimized for IA-32 and Intel® Itanium® architecture and running on Microsoft* Windows* and Linux* operating systems.

This manual provides detailed descriptions of Intel IPP functions developed for cryptographic operations.

This chapter introduces the Intel IPP cryptography software and explains the organization of this manual.

Basic Features

Like other members of Intel® Performance Libraries, Intel Integrated Performance Primitives is a collection of high-performance code that performs domain-specific operations. It is distinguished by providing a low-level, stateless interface.

Based on experience in developing and using Intel Performance Libraries, Intel IPP has the following major distinctive features:

- Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, image and video processing, operations on small matrices, and cryptography applications.

- Intel IPP functions follow the same interface conventions, including uniform naming conventions and similar composition of prototypes for primitives that refer to different application domains.
- Intel IPP functions use an abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up the performance, Intel IPP functions are optimized to use all benefits of Intel® architecture processors. Besides this, most of Intel IPP functions do not use complicated data structures, which helps reduce overall execution overhead.

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Itanium®-based platforms and systems with Intel® StrongARM* technology or Intel® XScale™ technology. For more information on platform compatibility, see Cross-Architecture Alignment. In addition, each Intel IPP function has its reference code written in ANSI C, which clearly presents the algorithm used and provides for compatibility with different operating systems.

About This Software

The Intel IPP software enables to take advantage of the single-instruction, multiple data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions.

The Intel Integrated Performance Primitives for cryptography provide a broad set of cryptographic primitive functions optimized for maximum performance on Intel® platforms, including the Intel® IA-32 and Intel® Itanium® architecture. The set of cryptography primitive functions can be thought of as a set of "building blocks" that enable independent software vendors to build their own FIPS-conformant security solutions.

The package of Intel IPP cryptography functions offers developers cross-platform and cross operating system API for routines commonly used for cryptographic operations. Use of Intel IPP primitive functions can help to drastically reduce development costs and accelerate time-to market by eliminating the need of writing processor-specific code for computation intensive routines.

Hardware and Software Requirements

Intel IPP for Intel architecture software runs on personal computers that are based on IA-32 processors or Itanium® architecture-based processors and running Microsoft Windows* 2000, Windows ME, Windows XP, or Linux*. Intel IPP integrates into the customer's application or library written in C or C++.

Platforms Supported

Intel IPP for Intel architecture software runs on Windows and Linux platforms. The code and syntax used in this manual for function and variable declarations are written in the ANSI C style. However, versions of Intel IPP for different processors or operating systems may, of necessity, vary slightly.

Cross-Architecture Alignment

Cross Architecture Overview

Intel IPP has been designed to support application development on various Intel® architectures. Previously, Intel IPP was offered in two separate products, one for the Intel® Pentium®, Xeon® and Itanium® processors and one for the Intel® PCA processors based on Intel XScale® technology. While these separate packages included many similarities, prior to version 4.0, there were some differences in both functionality and interface.

With rising interest in cross architecture development, the Intel IPP development teams have worked to bridge these differences to allow for easy development of applications for multiple Intel® platforms. Starting from version 4.0 onwards, Intel IPP represents the result of this effort and provides alignment of the two separate packages into a single offering that includes support for all of these architectures. This includes interface alignment and full API alignment for all functions that are relevant to both architectures.

With this release, the functions available for Intel Pentium, Xeon and Itanium processors represent a superset of functions available for the Intel PCA processors. This means that the API definition is common for all processors, while the underlying function implementation takes into account the variations in processor architectures.

By providing a single cross-architecture API, Intel IPP allows software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, mobile, and handheld platforms. Developers can write their code once in order to realize the application performance over many processor generations.

For additional information on API additions and changes from previous versions, please refer to the product release notes and the document *Migration Guide for Intel® IPP Cross Architecture API Alignment* available at http://www.intel.com/software/products/ipp.

The following table summarizes the functionality covered in each Intel IPP implementation.

Table 1-1 Function Coverage in Intel IPP

Function Group	Intel® Pentium® 4 processors	Intel® Xeon® processors with Intel® EM64T	Intel® Itanium® 2 processors	Intel® PCA processors	Intel® IXP4XX Product Line
Signal Processing	available	available	available	available[1]	available
Image Processing	available	available	available	available[1]	available
JPEG	available	available	available	available[1]	available
Speech Recognition	available	available	available	n/a	available
Speech Coding	available	available	available	available[1]	available
Audio Codecs	available	available	available	available[1]	available
Video Codecs	available	available	available	available[1]	available
Matrix	available	available	available	n/a	n/a
Vector Math	available	available	available	n/a	n/a
Computer Vision	available	available	available	n/a	available
Cryptography	available	available	available	available	available
Data Compression	available	available	available	n/a	available
Color Conversion	available	available	available	n/a	available
String Processing	available	available	available	n/a	available

1. Intel PCA processors support a subset of these functions.

API Changes in Version 5.0

As Intel IPP evolved and the functions got aligned for the Intel Pentium, Xeon and Itanium processors and the Intel PCA processors, the libraries accumulated numerous changes, which, in particular, caused quite a few functions grow obsolete. So, in Intel IPP 5.0, compatibility in API with previous versions had to be compromised to achieve the following library improvements:

- A number of functions were replaced by newer functions with extended functionality.
- Some other functions were changed to make the API more consistent.
- Odd and unusable functions were removed.

Several modifications of Intel IPP, and especially those that improve directory structure, also break *binary* backward compatibility of version 5.0. So, to migrate to Intel IPP 5.0, you have at least to rebuild your applications. If an application calls a function missing from Intel IPP 5.0, you have

also to modify the source code. To help you migrate to Intel IPP 5.0, Appendix A lists functions removed from Intel IPP 5.0 for cryptography and specifies Intel IPP 5.0 function(s) to substitute for the missing ones. Higher versions of Intel IPP, starting from 5.1, will be fully backward compatible with version 5.0.

Technical Support

Intel IPP provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, see http://developer.intel.com/software/products/.

Intel also provides a support web site that contains a rich repository of self-help information, including getting started tips, known product issues, product errata, license information, and more (visit http://support.intel.com/support/).

Registering your product entitles you to one-year technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, or contact Intel, or seek product support, please visit http://www.intel.com/software/products/support .

About This Manual

This manual provides a background for cryptography concepts used in the Intel IPP software as well as detailed description of the respective Intel IPP functions. The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter.

Manual Organization

This manual contains the following chapters and appendices:

Chapter 1 Overview. Introduces Intel IPP for cryptography operations, provides information on manual organization, and explains notational conventions.

Chapter 2 Symmetric Cryptography Primitive Functions. Explains basic concepts underlying the Intel IPP functions used for symmetric cryptography algorithm operations and describes the supported data layout and operation modes.

Chapter 3 One-Way Hash Primitives. Describes functions used for hash cryptography operations and data authentication.

Chapter 4 Data Authentication Primitive Functions. Describes functions for generating message authentication code using the Keyed Hash Functions (HMAC) and the Data authentication Functions (DAA) schemes.

Chapter 5 Public Key Cryptography Functions. Explains basic concepts underlying the Intel IPP functions used for asymmetric cryptography algorithm operations and describes the supported data layout and operation modes. In addition, the chapter covers the Diffie-Hellman functionality functions and the elliptic cryptography primitives operated on finite fields.

Appendix A Functions Removed from Intel® Integrated Performance Primitives 5.0. Lists cryptography functions removed from Intel IPP 5.0 along with their version 5.0 substitutes.

Appendix B Subsidiary Source Code Used in Examples. Presents source code of functions and classes used in code examples given in the manual chapters.

The manual also includes a Bibliography and Index of major terms and definitions used in this volume.

Function Descriptions

In Chapters 2 through 4, each function is introduced by its short name (without the `ipps` prefix and descriptors) and a brief description of its purpose. This is followed by an example of the function call sequence, definition of its parameters, and a more detailed explanation of the function purpose. The following sections are included in the function description:

Syntax	Lists function prototypes.
Parameters	Describes all function parameters.
Description	Defines the function and details the operation performed by the function. Code examples and equations that the function implements may be included in the description.
Return Value	Describes values indicating status codes set as the result of the function execution.

Audience for This Manual

This manual is intended for cryptography system developers who may benefit from avoiding hand optimization of cryptography operations. In the Intel IPP primitive functions for cryptography, developers may find a convenient and easy-to-use building block resource that covers cryptography algorithms widely used for creating products that require data security and integrity.

The audience must have experience using C and a working knowledge of the vocabulary and principles of basic crypto algorithms.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code
- Naming conventions for different items.

Font Conventions

The following font conventions are used throughout this manual:

`This type style`	Mixed with the uppercase in function names, code examples, and call statements, for example, `ippsAdd_BNU`.
`This type style`	Parameters in function prototype parameters and parameters description, for example, *`pCtx`*, *`pSrcMesg`*.

Naming Conventions

The naming conventions for different items are the same as used by the Intel IPP software.

- All names of the functions used for cryptographic operations have the `ipps` prefix. In code examples, you can distinguish the Intel IPP interface functions from the application functions by this prefix.

NOTE

NOTE. In this manual, the *`ipps`* prefix in function names is always used in code examples and function prototypes. In the text, this prefix is omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character, without underscore, for example, `ippsDESInit`.

Online Version

This manual is available in an electronic format (Portable Document Format, or PDF). To obtain a hard copy of the manual, print the file using the printing capability of Adobe Acrobat*, the tool used for the online presentation of the document.

Symmetric Cryptography Primitive Functions

2

In the context of secure data communication, symmetric cryptography primitive functions protect messages transferred over open communication media by offering adequate security strength to meet application security requirement, as well as algorithmic efficiency to enable secure communication in real-time.

Intel® Integrated Performance Primitives (Intel® IPP) for cryptography offer operations using the following symmetric cryptography algorithms:

- DES, Triple DES (TDES) [FIPS PUB 46-3], Rijndael [AES], Twofish [TF], and Blowfish [BF] block ciphers
- ARCFour stream cipher, described, for example, in [AC] and producing the same encryption/decryption as the RC4* proprietary cipher of RSA Security Inc.

The full list of functions is given in Table 2-1.

Table 2-1 Intel IPP Symmetric Algorithm Functions

Function Base Name	Operation
DES Functions	
`DESGetSize`	Gets the size of the `IppsDESSpec` context.
`DESInit`	Initializes user supplied memory as `IppsDESSpec` context for future use.
`DESEncryptECB`	Encrypts a variable length data stream in the ECB mode.
`DESDecryptECB`	Decrypts a variable length data stream in the ECB mode.
`DESEncryptCBC`	Encrypts a variable length data stream in the CBC mode.
`DESDecryptCBC`	Decrypts a variable length data stream in the CBC mode.
`DESEncryptCFB`	Encrypts a variable length data stream in the CFB mode.

Table 2-1 Intel IPP Symmetric Algorithm Functions (continued)

Function Base Name	Operation
DESDecryptCFB	Decrypts a variable length data stream in the CFB mode.
DESEncryptCTR	Encrypts a variable length data stream in the CTR mode.
DESDecryptCTR	Decrypts a variable length data stream in the CTR mode.
TDES Functions	
TDESEncryptECB	Encrypts variable length data stream in the ECB mode.
TDESDecryptECB	Decrypts variable length data stream in the ECB mode.
TDESEncryptCBC	Encrypts variable length data stream in the CBC mode.
TDESDecryptCBC	Decrypts variable length data stream in the CBC mode.
TDESEncryptCFB	Encrypts variable length data stream in the CFB mode.
TDESDecryptCFB	Decrypts variable length data stream in the CFB mode.
TDESEncryptCTR	Encrypts a variable data stream in the CTR mode.
TDESDecryptCTR	Decrypts a variable length data stream in the CTR mode.
Rijndael Algorithm Functions	
Rijndael128GetSize	Gets the size of the `IppsRijndael128Spec` context.
Rijndael128Init	Initializes user supplied memory as `IppsRijndael128Spec` context for future use.
Rijndael128EncryptECB	Encrypts plaintext message using the ECB encryption mode.
Rijndael128DecryptECB	Decrypts byte data stream using Rijndael algorithm in the ECB mode.
Rijndael128EncryptCBC	Encrypts byte data stream according to Rijndael in the CBC mode.
Rijndael128DecryptCBC	Decrypts byte data stream according to Rijndael in the CBC mode.
Rijndael128EncryptCFB	Encrypts byte data stream according to Rijndael in the CFB mode.
Rijndael128DecryptCFB	Decrypts byte data stream according to Rijndael in the CFB mode.
Rijndael128EncryptCTR	Encrypts a variable length data stream in the CTR mode.
Rijndael128DecryptCTR	Decrypts a variable length data stream in the CTR mode.
Rijndael128EncryptCCM	Encrypts a variable length data stream and generates its authentication tag in the CCM mode.
Rijndael128DecryptCCM	Decrypts and verifies a variable length data stream in the CCM mode.
Rijndael192GetSize	Gets the size of the `IppsRijndael192Spec` context.
Rijndael192Init	Initializes user supplied memory as `IppsRijndael192Spec` context for future use.

Table 2-1 Intel IPP Symmetric Algorithm Functions (continued)

Function Base Name	Operation
Rijndael192EncryptECB	Encrypts a byte data stream according to Rijndael in the ECB mode.
Rijndael192DecryptECB	Decrypts byte data stream according to Rijndael in the EBC mode.
Rijndael192EncryptCBC	Encrypts a byte data stream according to Rijndael in the CBC mode.
Rijndael192DecryptCBC	Decrypts a byte data stream according to Rijndael in the CBC mode.
Rijndael192EncryptCFB	Encrypts a byte data stream according to Rijndael in the CFB mode.
Rijndael192DecryptCFB	Decrypts a byte data stream according to Rijndael in the CFB mode.
Rijndael192EncryptCTR	Encrypts a variable length data stream in the CTR mode.
Rijndael192DecryptCTR	Decrypts a variable length data stream in the CTR mode.
Rijndael256GetSize	Gets the size of the `Rijndael256Spec` context.
Rijndael256Init	Initializes user supplied memory as `IppsRijndael256Spec` context for future use.
Rijndael256EncryptECB	Encrypts a byte data stream according to Rijndael in the ECB mode.
Rijndael256DecryptECB	Decrypts a byte data stream according to Rijndael in the ECB mode.
Rijndael256EncryptCBC	Encrypts byte data stream according to Rijndael in the CBC mode.
Rijndael256DecryptCBC	Decrypts byte data stream according to Rijndael in the CBC mode.
Rijndael256EncryptCFB	Encrypts byte data stream according to Rijndael in the CFB mode.
Rijndael256DecryptCFB	Decrypts byte data stream according to Rijndael in the CFB mode.
Rijndael256EncryptCTR	Encrypts a variable length data stream in the CTR mode.
Rijndael256DecryptCTR	Decrypts a variable length data stream in the CTR mode.
Blowfish Algorithm Functions	
BlowfishGetSize	Gets the size of the `IppsBlowfishSpec` context.
BlowfishInit	Initializes user supplied memory as `IppsBlowfishSpec` context for future use.
BlowfishEncryptECB	Encrypts input plaintext in the ECB mode.
BlowfishDecryptECB	Decrypts byte data stream according to Blowfish scheme in the EBC mode.

Table 2-1 Intel IPP Symmetric Algorithm Functions (continued)

Function Base Name	Operation
BlowfishEncryptCBC	Encrypts byte data stream according to Blowfish scheme in the CBC mode.
BlowfishDecryptCBC	Decrypts byte data stream according to Blowfish scheme in the CBC mode.
BlowfishEncryptCFB	Encrypts byte data stream according to Blowfish scheme in the CFB mode.
BlowfishDecryptCFB	Decrypts byte data stream according to Blowfish scheme in the CFB mode.
BlowfishEncryptCTR	Encrypts a variable length data stream in the CTR mode.
BlowfishDecryptCTR	Decrypts a variable length data stream in the CTR mode.
Twofish Algorithm Functions	
TwofishGetSize	Gets the size of the `IppsTwofishSpec` context.
TwofishInit	Initializes user supplied memory as `IppsTwofishSpec` context for future use.
TwofishEncryptECB	Encrypts input plaintext in the ECB mode.
TwofishDecryptECB	Decrypts byte data stream according to Twofish scheme in the EBC mode.
TwofishEncryptCBC	Encrypts byte data stream according to Twofish scheme in the CBC mode.
TwofishDecryptCBC	Decrypts byte data stream according to Twofish scheme in the CBC mode.
TwofishEncryptCFB	Encrypts byte data stream according to Twofish scheme in the CFB mode.
TwofishDecryptCFB	Decrypts byte data stream according to Twofish scheme in the CFB mode.
TwofishEncryptCTR	Encrypts a variable length data stream in the CTR mode.
TwofishDecryptCTR	Decrypts a variable length data stream in the CTR mode.
RCFour Algorithm Functions	
ARCFourGetSize	Gets the size of the `IppsARCFourState` context.
ARCFourCheckKey	Checks weakness of a user defined key.
ARCFourInit	Initializes user supplied memory as the `IppsARCFourState` context for future use.
ARCFourEncrypt	Encrypts a variable length data stream according to ARCFour.
ARCFourDecrypt	Decrypts a variable length data stream according to ARCFour.

Table 2-1 Intel IPP Symmetric Algorithm Functions (continued)

Function Base Name	Operation
`ARCFourReset`	Resets the `IppsARCFourState` context to the initial state.

Block Cipher Modes Operation

Most of Symmetric Cryptography Algorithms implemented in Intel IPP are Block Ciphers, which operate on data blocks of the fixed size. Block Ciphers encrypt a plaintext block into a ciphertext block or decrypts a ciphertext block into a plaintext block. The size of the data blocks depends on the specific algorithm. Table 2-2 shows the correspondence between Block Ciphers applied and their data block size.

Table 2-2 Block Sizes in Symmetric Algorithms

Block Cipher Name	Data Block Size (bits)
DES	64
TDES	64
Rijndael128	128
Rijndael192	192
Rijndael256	256
Twofish	128
Blowfish	64

Block Cipher modes of executing the operation of encryption/decryption are applied in practice more frequently than "pure" Block Ciphers. On one hand, the modes enable you to process arbitrary length data stream. On the other hand, they provide additional security strength.

Intel IPP for cryptography supports four widely used modes, as specified in [NIST SP 800-38A]:

- Electronic Code Book (ECB) mode
- Cipher Block Chain (CBC) mode
- Cipher Feedback (CFB) mode
- Counter (CTR) mode.

NOTE. For simplicity and consistency, the mathematical expression and pseudo code in this chapter describes the behaviour of each function.

The cryptographic functions described in this chapter require the application to specify both the plaintext message and the ciphertext message lengths as multiples of block size of the respective algorithm (see Table 2-2). To meet this requirement in ciphering the message, the application may use any padding scheme, for example, the scheme defined in [PKCS7]. In case padding is used, the application is responsible for correct interpretation and processing of the last deciphered message block. So of the three padding schemes available for the previous release

```
typedef enum {
    NONE  = 0, IppsCPPaddingNONE  = 0,
    PKCS7 = 1, IppsCPPaddingPKCS7 = 1,
    ZEROS = 2, IppsCPPaddingZEROS = 2
} IppsCPPadding;
```

only `IppsCPPaddingNONE` remains acceptable.

DES/TDES Functions

Data Encryption Standard (DES) is a well-known symmetric cipher and also the first modern commercial-grade algorithm with open and fully specified implementation details. DES consists of a Feistel network iterated 16 times with the block size of 64 bits and the effective key size of 56 bits.

Triple Data Encryption Standard (TDES) is a revised symmetric algorithm scheme built on the DES system. TDES encryption process includes three consecutive DES operations in the encryption, decryption, and encryption (E-D-E) sequence again in accordance with the American standard FIPS 46-3.

Although the functions that support TDES operations require three sets of round keys, the functions can operate under TDES cipher system with a two-set round keys by simply setting the third set of round keys to be the same as the first set.

You can use the functions described in this section for performing various operational modes under the DES/TRES cipher systems.

NOTE. Intel IPP functions for cryptography operations do not allocate memory internally. The function `GetSize` does not require allocated memory. You need to call the function `GetSize` to find out how much available memory you need to have to work with the selected algorithm and after that you call the initialization function to create a memory buffer and initialize it.

Intel IPP for cryptography supports ECB, CBC, CFB, and CTR modes. You can tell which algorithm a given function supports from the function base name, for example, the function `DESEncryptECB` operates under the ECB mode for DES encryption and the function `TDESEncryptECB` operates under the ECB mode under the TDES scheme.

The encryption functions `DESEncryptCBC` and `TDESEncryptCBC` operate under the CBC mode using their respective cipher scheme and require to have an initialization vector *`iv`*. Since there exists a number of ways to initialize the initialization vector *`iv`*, you should remember which of these ways you used to be able to decrypt the message when needed.

Functions `DESEncryptCFB` and `TDESEncryptCFB` operate under CFB mode for encryption using their respective cipher scheme, both require having the initialization vector *`pIV`*, and CFB block size *`cfbBlkSize`*.

All functions described in this section use the context `IppsDESSpec` to serve as an operational vehicle that carries a set of round keys.

The application code for conducting a typical encryption under CBC mode using the TDES scheme must perform the following sequence of operations:

1. Get the size required to configure the context `IppsDESSpec` by calling the function `DESGetSize`.
2. Call operating system memory allocation service function to allocate three buffers whose sizes are not less than the one specified by the function `DESGetSize`. Initialize pointers to contexts *`pCtx1`*, *`pCtx2`*, and *`pCtx3`* by calling the function `DESInit` three times, each with the allocated buffer and the respective DES key.
3. Specify the initialization vector and then call the function `TDESEncryptCBC` to encrypt the input data stream under CBC mode using TDES scheme.

4. Free the memory allocated to the buffer once TDES encryption under the CBC mode has been completed and the data structures allocated for set of round keys are no longer required.

> **NOTE.** Similar procedure can be applied for ECB, CFB, and CTR mode operation.

DESGetSize

Gets the size of the `IppsDESSpec` *context.*

Syntax

```
IppStatus ippsDESGetSize(int* pSize)
```

Parameters

pSize	Pointer to the `IppsDESSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsDESSpec` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DESInit

Initializes user supplied memory as the `IppsDESSpec` *context for future use.*

Syntax

```
IppStatus ippsDESInit(const Ipp8u* pKey, IppsDESSpec* pCtx);
```

Parameters

pKey	Pointer to the DES key.
pCtx	Pointer to the `IppsDESSpec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsDESSpec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DESEncryptECB

Encrypts a variable length data stream in the ECB mode.

Syntax

```
IppStatus ippsDESEncryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsDESSpec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.

pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `DECSpec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DESDecryptECB

Decrypts a variable length data stream in the ECB mode.

Syntax

```
IppStatus ippsDESDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsDESSpec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream of variable length.
pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext data stream in bytes.

pCtx	Pointer to the `IppsDESSpec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

DESEncryptCBC

Encrypts a variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsDESEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsDESSpec* pCtx, Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the plaintext data stream length in bytes.
pCtx	Pointer to the `IppsDESSpec` context.
pIV	Pointer to the initialization vector for the CBC mode operation.

padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCBC

Decrypts a variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsDESDecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsDESSpec* pCtx, Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsDESSpec` context.
pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.

DESEncryptCFB

Encrypts a variable length data stream in the CFB mode.

Syntax

```
IppStatus ippsDESEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int
    cfbBlkSize, const IppsDESSpec* pCtx, Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srcLen`	Length of the plaintext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsDESSpec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by *`cfbBlkSize`* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCFB

Decrypts a variable length data stream in the CFB mode.

Syntax

```
IppStatus ippsDESDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    int cfbBlkSize, const IppsDESSpec* pCtx, const Ipp8u* pIV,
    IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsDESSpec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.

padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by cipher block size.

DESEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsDESEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsDESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Pointer to the input plaintext data stream of a variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `IppsDESSpec` context.
pCtrValue	Pointer to the counter data block.

ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *crtNumBitSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsDESDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsDESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of a variable length.
srclen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `IppsDESSpec` context.
pCtrValue	Pointer to the counter data block.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESEncryptECB

Encrypts variable length data stream in ECB mode.

Syntax

```
IppStatus ippsTDESEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, IppsCPPadding padding);
```

Parameters

`pSrc`	Input plaintext data stream of a variable length.
`pDst`	Resulting ciphertext data stream.
`srclen`	Input data stream length in bytes.
`pCtx1`	First set of round keys scheduled for TDES internal operations.
`pCtx2`	Second set of round keys scheduled for TDES internal operations.
`pCtx3`	Third set of round keys scheduled for TDES internal operations.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of supplied round keys in the ECB mode. The function returns the ciphertext result.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if the input data stream length is not divisible by cipher block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptECB

Decrypts variable length data stream in the ECB mode.

Syntax

```
IppStatus ippsTDESDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, IppsCPPadding padding);
```

Parameters

pSrc	Input ciphertext data stream of variable length.
pDst	Resulting plaintext data stream.
srclen	Input data stream length in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.
pCtx3	Third set of round keys scheduled for TDES internal operations.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of supplied round keys in the ECB mode. The function returns the ciphertext result and validates the final plaintext block.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.

TDESEncryptCBC

Encrypts variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsTDESEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Input plaintext data stream of a variable length.
`pDst`	Resulting ciphertext data stream.
`pIV`	Initialization vector for TDES CBC mode operation.
`srclen`	Input data stream length in bytes.
`pCtx1`	First set of round keys scheduled for TDES internal operations.
`pCtx2`	Second set of round keys scheduled for TDES internal operations.
`pCtx3`	Third set of round keys scheduled for TDES internal operations.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Block Chaining (CBC) mode with the initialization vector. The function returns the ciphertext result.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if the input data stream length is not divisible by cipher block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCBC

Decrypts variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsTDESDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

pSrc	Input ciphertext data stream of a variable length.
pDst	Resulting plaintext data stream.
pIV	Initialization vector for TDES CBC mode operation.
srclen	Input data stream length in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.

pCtx3	Third set of round keys scheduled for TDES internal operations.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Block Chaining (CBC) mode with the initialization vector. The function returns the ciphertext result and validates the final plaintext block.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

TDESEncryptCFB

Encrypts variable length data stream in the CFB mode.

Syntax

```
IppStatus ippsTDESEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    int cfbBlkSize, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2,
    const IppsDESSpec *pCtx3, Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

pSrc	Input plaintext data stream of variable length.
pDst	Resulting ciphertext data stream.
pIV	Initialization vector for TDES CFB mode operation.

srclen	Input data stream length in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.
pCtx3	Third set of round keys scheduled for TDES internal operations.
cfbBlkSize	CFB block size in bytes.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Feedback (CFB) mode with the initialization vector. The function returns the ciphertext result.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by *cfbBlkSize* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCFB

Decrypts variable length data stream in the CFB mode.

Syntax

```
IppStatus ippsTDESDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    int cfbBlkSize, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2,
    const IppsDESSpec *pCtx3, Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

pSrc	Input ciphertext data stream of variable length.
pDst	Resulting plaintext data stream.
pIV	Initialization vector for TDES CFB mode operation.
srclen	Ciphertext data stream length in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.
pCtx3	Third set of round keys scheduled for TDES internal operations.
cfbBlkSize	CFB block size in bytes.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Feedback (CFB) mode with the initialization vector. The function returns the ciphertext result and validates the final plaintext block.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by cipher block size.

TDESEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsTDESEncryptCTR(const Ipp8u *pSrc, Ipp8u *pDst, int srclen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, Ipp8u *pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Input plaintext data stream of a variable length.
pDst	Resulting ciphertext data stream.
srclen	Input data stream length in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.
pCtx3	Third set of round keys scheduled for TDES internal operations.
pCtrValue	Counter.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function `TDESEncryptCTR` encrypts the input data stream of a variable length according to the cipher scheme specified in the [NIST SP 800-38A] recommendation. The function uses three sets of the supplied round keys. The standard incrementing function is applied to increment counter value. The function returns the ciphertext result.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less that or equal to zero.

`ippStsCTRSizeErr`	Indicates an error condition if the value of the *crtNumBitSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsTDESDecryptCTR(const Ipp8u *pSrc, Ipp8u *pDst, int srcLen,
    const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec
    *pCtx3, Ipp8u *pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Input ciphertext data stream of a variable length.
pDst	Resulting plaintext data stream.
srcLen	Length of the plaintext data stream in bytes.
pCtx1	First set of round keys scheduled for TDES internal operations.
pCtx2	Second set of round keys scheduled for TDES internal operations.
pCtx3	Third set of round keys scheduled for TDES internal operations.
pCtrValue	Counter.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function `TDESDecryptCTR` decrypts the input data stream of a variable length according to the cipher scheme specified in the [NIST SP 800-38A] recommendation. The function uses three sets of the supplied round keys. The standard incrementing function is applied to increment value of counter. The function returns the ciphertext result.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the descrypted plaintext data stream length is less that or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Example 2-1 DES/TDES Encryption and Decryption

```
   // use of the ECB mode
void DES_sample(void){
   // size of the DES algorithm block is equal to 8
   const int desBlkSize = 8;

   // get size of the context needed for the encryption/decryption operation
   int ctxSize;
   ippsDESGetSize(&ctxSize);
   // and allocate one
   IppsDESSpec* pCtx = (IppsDESSpec*)( new Ipp8u [ctxSize] );

   // define the key
   Ipp8u key[] = {0x01,0x2,0x3,0x4,0x5,0x6,0x7,0x8};
   // and prepare the context for the DES usage
   ippsDESInit(key, pCtx);
   // define the message to be encrypted
   Ipp8u ptext[] = {"quick brown fox jupm over lazy dog"};

   // allocate enough memory for the ciphertext
   // note that
   // the size of ciphertext is always multiple of cipher block size
```

Example 2-1 DES/TDES Encryption and Decryption (continued)

```
   Ipp8u ctext[(sizeof(ptext)+desBlkSize-1) &~(desBlkSize-1)];
   // encrypt (ECB mode) ptext message
   // pay attention to the 'length' parameter
   // it defines the number of bytes to be encrypted
   ippsDESEncryptECB(ptext, ctext, sizeof(ptext),
                     pCtx,
                     IppsCPPaddingZEROS);
      // allocate memory for the decrypted message
   Ipp8u rtext[sizeof(ptext)];
   // decrypt (ECB mode) ctext message
   // pay attention to the 'length' parameter
   // it defines the number of bytes to be decrypted
   ippsDESDecryptECB(ctext, rtext, sizeof(ptext),
                     pCtx,
                     IppsCPPaddingZEROS);
   delete (Ipp8u*)pCtx;
}
```

Rijndael Functions

Rijndael cipher scheme is an iterated block cipher with a variable block size and a variable key length. You can independently specify the lengths of the data block and the key as 128, 192, or 256 bits.

This section describes the functions operating in various operational modes under the various Rijndael cipher systems. The functions in this section are categorized by their data block sizes of the baseline Rijndael cipher functions:

- `Rijndael128` refers to the Rijndael cipher scheme with 128-bit data block size
- `Rijndael192` refers to the Rijndael cipher scheme with 192-bit data block size
- `Rijndael256` refers to the Rijndael cipher scheme with 256-bit data block size.

To specify the key length for these baseline Rijndael cipher schemes, all the functions in this section use the following enumeration

```
typedef enum {
IppsRijndaelKey128 = 128, // 128-bit key
IppsRijndaelKey192 = 192, // 192-bit key
IppsRijndaelKey256 = 256, // 256-bit key
} IppsRijndaelKeyLength;
```

The functions for Rijndael128 with the 128-bit key length described in this section are, in fact, American Encryption Standard (AES) cipher functions implemented in the way to comply with the American Standard FIPS 197. All other functions for various other Rijndael block cipher schemes fully comply to the respective cipher schemes documented by Joan Daeman and Vincent Rijmen.

Throughout this section, the functions for Rijndael128 baseline cipher schemes employ the context `IppsRijndael128Spec` and the functions for Rijndael256 baseline cipher schemes employ the context `IppsRijndael256Spec`. They are defined to serve as operational vehicles not only to carry a set of round keys and a set of round inverse keys at the same time, but also the key management information.

Once the respective initialization function generates the round keys, the functions for ECB, CBC, CFB, and other modes are ready for the execution of either encrypting or decrypting the streaming data with the specified padding scheme.

The application code for conducting a typical encryption under CBC mode using the AES scheme, that is, the Rijndael128 with a 128-bit key, should follow the sequence of operations as outlined below:

1. Get the size required to configure the context `IppsRijndael128Spec` by calling the function `Rijndael128GetSize`.
2. Call the operating system memory-allocation service function to allocate a buffer whose size is no less than the one specified by the function `Rijndael128GetSize`.
3. Initialize the context `IppsRijndael128Spec` *`*pCtx`* by calling the function `Rijndael128Init` with the allocated buffer and the respective 128-bit AES key.
4. Specify the initialization vector and the padding scheme, then call the function `Rijndael128EncryptCBC` to encrypt the input data stream using the AES encryption function with CBC mode.
5. Call the operating system memory free service function to release the buffer allocated for the context `IppsRijndael128Spec`, if needed.

Rijndael128GetSize

Gets the size of the `IppsRijndael128Spec` *context.*

Syntax

```
IppStatus ippsRijndael128GetSize(int* pSize)
```

Parameters

pSize	Pointer to the `IppsRijndael128Spec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael128Spec` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Rijndael128Init

Initializes user supplied memory as `IppsRijndael128Spec` *context for future use.*

Syntax

```
IppStatus ippsRijndael128Init(const Ipp8u *pKey, IppsRijndaelKeyLength
    keylen, IppsRijndael128Spec* pCtx);
```

Parameters

pKey	Pointer to the Rijndael128 key.
keylen	Key byte stream length in bytes defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsRijndael128Spec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsRijndael128Spec`. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Returns an error condition if *keyLen* is not `IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`.

Rijndael128EncryptECB

Encrypts plaintext message by using ECB encryption mode.

Syntax

```
IppStatus ippsRijndael128EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
    srclen, const IppsRijndael128Spec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the input plaintext data in bytes.
pCtx	Pointer to the `IppsRijndael128Spec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptECB

Decrypts byte data stream by using Rijndael algorithm in the ECB mode.

Syntax

```
IppStatus ippsRijndael128DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael128Spec* pCtx, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream of variable length.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.

Rijndael128EncryptCBC

Encrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael128EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsRijndael128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srclen`	Length of the plaintext data stream length in bytes.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.
`pIV`	Pointer to the initialization vector for the CBC mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCBC

Decrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael128DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael128Spec* pCtx, const Ipp8u* pIV,
    IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsRijndael128Spec` context.
pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

Rijndael128EncryptCFB

Encrypts byte data stream according to Rijndael in the CFB mode.

```
IppStatus ippsRijndael128EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, int cfbBlkSize, const IppsRijndael128Spec* pCtx, const Ipp8u
    *pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.
pCtx	Pointer to the `IppsRijndael128Spec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by *`cfbBlkSize`* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCFB

Decrypts byte data stream according to Rijndael in CFB mode.

Syntax

```
IppStatus ippsRijndael128DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen,  int cfbBlkSize, const IppsRijndael128Spec* pCtx, const
    Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by cipher block size.

Rijndael128EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael128EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsRijndael128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of a variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael128DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsRijndael128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of a variable length.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128EncryptCCM

Encrypts a variable length data stream and generates its authentication tag in the CCM mode.

Syntax

```
IppStatus ippsRijndael128EncryptCCM(const Ipp8u* pNonce, int nonceLen,
    const Ipp8u* pAssc, int asscLen, const Ipp8u* pSrc, int srcLen, int
    macLen, Ipp8u* pDst, const IppsRijndael128Spec* pCtx);
```

Parameters

`pNonce`	Pointer to the nonce.
`nonceLen`	Length of the nonce **`pNonce`* (in octets).
`pAssc`	Pointer to the assosiated data.
`asscLen`	Length of the assosiated data **`pAssc`* (in octets).
`pSrc`	Pointer to the input plaintext data.
`srcLen`	Length of the input plaintext **`pSrc`* (in octets).
`macLen`	Length of the authentication tag in octets.
`pDst`	Pointer to the resulting ciphertext data stream.
`pCtx`	Pointer to the `IppsRijndael128Spec` context.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length and computes the authentication tag according to the Counter with Cipher Block Chainng-Message Authentication Code (CCM) mode, as specified in [NIST SP 800-38C].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \le noncelen \le 13$ $asscLen \ge 0$ $scrLen > 0$ $macLen = 2 \cdot n, (1 \le n \le 8)$.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCCM

Decrypts and verifies a variable length data stream in the CCM mode.

Syntax

```
IppStatus ippsRijndael128DecryptCCM(const Ipp8u* pNonce, int nonceLen, const
    Ipp8u* pAssc, int asscLen, const Ipp8u* pSrc, int srcLen, int macLen, Ipp8u*
    pDst, IppBool* pResult, const IppsRijndael128Spec* pCtx);
```

Parameters

pNonce	Pointer to the nonce of length.
nonceLen	Length of the nonce **pNonce* (in octets).
pAssc	Pointer to the assosiated data.

asscLen	Length of the assosiated data **pAssc* (in octets).
pSrc	Pointer to the input ciphertext data.
srcLen	Length of the input ciphertext **pSrc* (in octets).
macLen	Length of the authentication tag in octets.
pDst	Pointer to the resulting plaintext data stream.
pResult	Pointer to the result of verification.
pCtx	Pointer to the `IppsRijndael128Spec` context.

Description

This function is declared in the ippcp.h file. The function decrypts the input data stream of a variable length and verifies the authentication tag according to the Counter with Cipher Block Chainng-Message Authentication Code (CCM) mode, as specified in [NIST SP 800-38C].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \le noncelen \le 13$ $asscLen \ge 0$ $scrLen \ge maclen$ $macLen = 2 \cdot n, (1 \le n \le 8)$.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael192GetSize

Gets the size of the `IppsRijndael192Spec` *context.*

Syntax

```
IppStatus ippsRijndael192GetSize(int* pSize)
```

Parameters

pSize	Pointer to the `IppsRijndael192Spec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael192Spec` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Rijndael192Init

Initializes user supplied memory as the `IppsRijndael192Spec` *context for future use.*

Syntax

```
IppStatus ippsRijndael192Init(const Ipp8u *pKey, IppsRijndaelKeyLength
    keylen, IppsRijndael192Spec* pCtx);
```

Parameters

pKey	Pointer to the Rijndael192 key.
keylen	Key byte stream length in bytes as defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsRijndael192Spec` context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsRijndael192Spec` context. In addition, the function uses the key to provide all the necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keylen* is not set to `IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`.

Rijndael192EncryptECB

Encrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippsRijndael192EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
    srclen, const IppsRijndael192Spec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the input data stream in bytes.
pCtx	Pointer to the `IppsRijndael192Spec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

`ippStsNoErr`	Indicates no error. Any other value indocates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to 0.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptECB

Decrypts byte data stream according to Rijndael in the EBC mode.

Syntax

```
IppStatus ippsRijndael192DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael192Spec *pCtx, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream of variable length.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael192Spec` context.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.

Rijndael192EncryptCBC

Encrypts a byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael192EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsRijndael192Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srclen`	Length of the plaintext data stream length in bytes.
`pCtx`	Pointer to the `IppsRijndael192Spec` context.
`pIV`	Pointer to the initialization vector for the CBC mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by data block size.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCBC

Decrypts a byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael192DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael192Spec* pCtx, const Ipp8u* pIV,
    IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsRijndael192Spec` context.
pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

Rijndael192EncryptCFB

Encrypts a byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippsRijndael192EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, int cfbBlkSize, const IppsRijndael192Spec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.
pCtx	Pointer to the `IppsRijndael192Spec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.

`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by *`cfbBlkSize`* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCFB

Decrypts a byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippsRijndael192DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, int cfbBlkSize, const IppsRijndael192Spec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsRijndael192Spec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by cipher block size.

Rijndael192EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael192EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, const IppsRijndael192Spec* pCtx, Ipp8u* pCtrValue, int
    ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of a variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael192Spec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael192DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsRijndael192Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of a variable length.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael192Spec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael256GetSize

Gets the size of the `IppsRijndael256Spec` *context.*

Syntax

```
IppStatus ippsRijndael256GetSize(int* pSize)
```

Parameters

`pSize`	Pointer to the `IppsRijndael256Spec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael256Spec` context size in bytes and stores it in *`*pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Rijndael256Init

Initializes user supplied memory as `IppsRijndael256Spec` *context for future use.*

Syntax

```
IppStatus ippsRijndael256Init(const Ipp8u *pKey, IppsRijndaelKeyLength
    keylen, IppsRijndael256Spec* pCtx);
```

Parameters

pKey	Pointer to the Rijndael256 key.
keylen	Key byte stream length in bytes defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsRijndael256Spec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsRijndael256Spec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keylen* is not set to `IppsRijndaelKey128, IppsRijndaelKey192` or `IppsRijndaelKey256.`

Rijndael256EncryptECB

Encrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippsRijndael256EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
    srclen, const IppsRijndael256Spec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the input data stream in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

`ippStsNoErr`	Indicates no error. Any other value indocates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to 0.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptECB

Decrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippsRijndael256DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael256Spec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream of variable length.
pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext data stream in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

Rijndael256EncryptCBC

Encrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael256EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsRijndael256Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the plaintext data stream length in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
pIV	Pointer to the initialization vector for the CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCBC

Decrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippsRijndael256DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsRijndael256Spec* pCtx, const Ipp8u* pIV,
    IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

Rijndael256EncryptCFB

Encrypts byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippsRijndael256EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, int cfbBlkSize, const IppsRijndael256Spec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by *cfbBlkSize* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCFB

Decrypts byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippsRijndael256DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, int cfbBlkSize, const IppsRijndael256Spec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by cipher block size.

Rijndael256EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael256EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsRijndael256Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Pointer to the input plaintext data stream of a variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `IppsRijndael256Spec` context.
pCtrValue	Pointer to the counter data block.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.

`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael256DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsRijndael256Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of a variable length.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsRijndael256Spec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.

`ippStsCTRSizeErr` Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.

`ippStsContextMatchErr` Indicates an error condition if the context parameter does not match the operation.

Example 2-2 AES Encryption and Decryption

```
    // use of the CTR mode
void AES_sample(void){
   // size of Rijndael-128 algorithm block is equal to 16
   const int aesBlkSize = 16;

   // get the size of the context needed for the encryption/decryption operation
   int ctxSize;
   ippsRijndael128GetSize(&ctxSize);

   // and allocate one
   IppsRijndael128Spec* pCtx = (IppsRijndael128Spec*)( new Ipp8u [ctxSize] );
   // define the key
   Ipp8u key[16] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                    0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15};

   // and prepare the context for Rijndael128 usage
   ippsRijndael128Init(key,IppsRijndaelKey128, pCtx);

   // define the message to be encrypted
   Ipp8u ptext[] = {"quick brown fox jupm over lazy dog"};
```

Example 2-2 AES Encryption and Decryption (continued)

```
// define an initial vector
Ipp8u crt0[aesBlkSize] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
                          0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
Ipp8u  crt[aesBlkSize];

// counter the variable number of bits
int ctrNumBitSize = 64;

// allocate enough memory for the ciphertext
// note that
// the size of the ciphertext is always equal to that of the planetext
Ipp8u ctext[sizeof(ptext)];

// init the counter
memcpy(crt, crt0, sizeof(crt0));
// encrypt (CTR mode) ptext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be encrypted
memcpy(crt, crt0, sizeof(crt0));
ippsRijndael128EncryptCTR(ptext, ctext, sizeof(ptext),
                          pCtx,
                          crt, ctrNumBitSize);

// allocate memory for the decrypted message
Ipp8u rtext[sizeof(ptext)];

// init the counter
memcpy(crt, crt0, sizeof(crt0));
```

Example 2-2 AES Encryption and Decryption (continued)

```
   // decrypt (ECTR mode) ctext message
   // pay attention to the 'length' parameter
   // it defines the number of bytes to be decrypted
   ippsRijndael128DecryptCTR(ctext, rtext, sizeof(ptext),
                             pCtx,
                             crt, ctrNumBitSize);
   delete (Ipp8u*)pCtx;
}
```

Blowfish Functions

Blowfish is a 16-round Feistel block cipher. Under this algorithm, the block size is 64 bits and the key can be of any size up to 448 bits. Although the algorithm requires a computationally intensive key expansion process that creates a set of eighteen 32-bit subkeys plus four 8x32-bit S-boxes derived from the input key, for a total of 4168 bytes, the actual encryption of streaming data is very efficient for software implementation.

This section describes the functions performing various operational modes under the Blowfish cipher systems. The implementation of the functions described in this section complies with the Blowfish cipher schemes.

Throughout this section, the functions for Blowfish baseline cipher scheme employ the context `IppsBlowfishSpec`. It is so defined as to serve as the operational vehicles to not only carry both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the respective initialization function has generated a set of subkeys and S-Boxes, the functions for ECB, CBC, CFB, and CTR modes are ready to the execution of either encrypting or decrypting the streaming data with the selected padding scheme.

The application code for conducting a typical encryption under CBC mode using Blowfish scheme should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the context `IppsBlowfishSpec` by calling the function `BlowfishGetSize`.
2. Call the operating system memory allocation service function to allocate a buffer whose size is no less than the one specified by the function `BlowfishGetSize`. Initialize the context `IppsBlowfishSpec *pCtx` by calling the function `BlowfishInit` with the allocated buffer and the respective Blowfish cipher key of the specified size.
3. Specify the initialization vector and the padding scheme, then call the function `BlowfishEncryptCBC` to encrypt the input data stream using the Blowfish encryption function with CBC mode.
4. Call the operating system memory free service function to release the buffer allocated for the context `IppsBlowfishSpec`, if needed.

BlowfishGetSize

Gets the size of the `IppsBlowfishSpec` *context.*

Syntax

```
IppStatus ippsBlowfishGetSize(int* pSize)
```

Parameters

pSize	Pointer to the `IppsBlowfishSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsBlowfishSpec` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

BlowfishInit

Initializes user supplied memory as the `IppsBlowfishSpec` *context for future use.*

Syntax

```
IppStatus ippsBlowfishInit(const Ipp8u *pKey, int keylen,
    IppsBlowfishSpec* pCtx);
```

Parameters

pKey	Pointer to the Blowfish key.
keylen	Key byte stream length in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsBlowfishSpec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keylen* is less then 1 or more than 56.

BlowfishEncryptECB

Encrypts input plaintext in the ECB mode.

Syntax

```
IppStatus ippsBlowfishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
    scrlen, const IppsBlowfishSpec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the input data stream in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indocates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to 0.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptECB

Decrypts byte data stream according to Blowfish scheme in the EBC mode.

Syntax

```
IppStatus ippsBlowfishDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsBlowfishSpec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream of variable length.

pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext data stream in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

BlowfishEncryptCBC

Encrypts byte data stream according to Blowfish scheme in the CBC mode.

Syntax

```
IppStatus ippsBlowfishEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
   const IppsBlowfishSpec* pCtx, const Ipp8u* pIV,  IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the plaintext data stream length in bytes.

pCtx	Pointer to the `IppsBlowfishSpec` context.
pIV	Pointer to the initialization vector for the CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by data block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCBC

Decrypts byte data stream according to Blowfish scheme in the CBC mode.

Syntax

```
IppStatus ippsBlowfishDecryptCBC(const  Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsBlowfishSpec* pCtx, const Ipp8u* pIV,  IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.

pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

BlowfishEncryptCFB

Encrypts byte data stream according to Blowfish scheme in the CFB mode.

Syntax

```
IppStatus ippsBlowfishEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, int cfbBlkSize, const IppsBlowfishSpec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.

pCtx	Pointer to the `IppsBlowfishSpec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by *cfbBlkSize* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCFB

Decrypts byte data stream according to Blowfish scheme in the CFB mode.

Syntax

```
IppStatus ippsBlowfishDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, int cfbBlkSize, const IppsBlowfishSpec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of variable length.

srclen	Length of the ciphertext data stream in bytes.
cfbBlkSize	Size of the CFB block in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.
pIV	Pointer to the initialization vector for the CFB mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *cfbBlkSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *srcLen* is not divisible by cipher block size.

BlowfishEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsBlowfishEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsBlowfishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Pointer to the input plaintext data stream of a variable length.

pDst	Pointer to the resulting ciphertext data stream.
srcLen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.
pCtrValue	Pointer to the counter data block.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *crtNumBitSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsBlowfishDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsBlowfishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of a variable length.

srcLen	Length of the plaintext data stream in bytes.
pCtx	Pointer to the `IppsBlowfishSpec` context.
pCtrValue	Pointer to the counter data block.
ctrNumBitSize	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *crtNumBitSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Example 2-3 Blowfish Encryption and Decryption

```
   // use of the CBC mode
void BF_sample(void){
   // size of Blowfish algorithm block is equal to 8
   const int bfBlkSize = 8;

   // get the size of the context needed for the encryption/decryption operation
   int ctxSize;
   ippsBlowfishGetSize(&ctxSize);

   // and allocate one
   IppsBlowfishSpec* pCtx = (IppsBlowfishSpec*)( new Ipp8u [ctxSize] );
  // define the key
   // note that the key length may vary from 1 to 56 bytes
   Ipp8u key[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                  0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15,
                  0x16,0x17};
   // and prepare the context for Blowfish usage
   ippsBlowfishInit(key,sizeof(key), pCtx);

   // define the message to be encrypted
   Ipp8u ptext[] = {"quick brown fox jupm over lazy dog"};

   // define an initial vector
   Ipp8u iv[bfBlkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
```

Example 2-3 Blowfish Encryption and Decryption (continued)

```
    // allocate enough memory for the ciphertext
   // note that
   // the size of the ciphertext is always multiple of the cipher block size
   Ipp8u ctext[(sizeof(ptext)+bfBlkSize-1) &~(bfBlkSize-1)];

   // encrypt (CBC mode) ptext message
   // pay attention to the 'length' parameter
   // it defines  the number of bytes to be encrypted
   ippsBlowfishEncryptCBC(ptext, ctext, sizeof(ptext),
                          pCtx,
                          iv,
                          IppsCPPaddingPKCS7);
// allocate memory for the decrypted message
   Ipp8u rtext[sizeof(ptext)];
   // decrypt (CBC mode) ctext message
   // pay attention to the 'length' parameter
   // it defines the number of bytes to be decrypted
   ippsBlowfishDecryptCBC(ctext, rtext, sizeof(ptext),
                          pCtx,
                          iv,
                          IppsCPPaddingPKCS7);
   delete (Ipp8u*)pCtx;
}
```

Twofish Functions

Twofish is a 16-round Feistel block cipher. The block size is 128 bits and the key can be any size up to 256 bits. The algorithm is one of the five Advanced Encryption Standard (AES) finalists. The cipher design for both the round function and key schedule enables an efficient implementation in software. Even though Twofish is free and not patented, it is nevertheless efficient and highly secure block cipher.

This section describes the functions for various operational modes under the Twofish cipher systems. The functions in this section are implemented to comply to the Twofish cipher schemes documented and submitted to NIST for candidacy of AES by B. Schneier.

Throughout this section, the functions for Twofish baseline cipher scheme employ the context `IppsTwofishSpec`. This structure is defined to serve as an operational vehicle to not only carry both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the respective initialization function has generated a set of subkeys and S-Boxes, the functions for ECB, CBC, CFB, and CTR modes are ready to the execution of either encrypting or decrypting the streaming data with the selected padding scheme.

The application code for conducting typical encryption under the CBC mode using the Twofish scheme should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the context `IppsTwofishSpec` by calling the function `TwofishGetSize`.
2. Call operating system memory allocation service function to allocate a buffer whose size is no less than the one specified by the function `TwofishGetSize`. Initialize the context `IppsTwofishSpec` *`*pCtx`* by calling the function `TwofishInit` with the allocated buffer and the respective Twofish cipher key of the specified size.
3. Specify the initialization vector and the padding scheme, than call the function `TwofishEncryptCBC` to encrypt the input data stream using the Twofish encryption function with CBC mode.
4. Call the operating system memory free service function to release the buffer allocated for the context `IppsTwofishSpec`, if needed.

TwofishGetSize

Gets the size of the `IppsTwofishSpec` *context.*

Syntax

```
IppStatus ippsTwofishGetSize(int* pSize)
```

Parameters

pSize	Pointer to the `IppsTwofishSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsTwofishSpec` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

TwofishInit

Initializes user supplied memory as the `IppsTwofishSpec` *context for future use.*

Syntax

```
IppStatus ippsTwofishInit(const Ipp8u *pKey, int keylen,
    IppsTwofishSpec* pCtx);
```

Parameters

pKey	Pointer to the Twofish key.
keylen	Key byte stream length in bytes.
pCtx	Pointer to the `IppsTwofishSpec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsTwofishSpec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keylen`* is less than 1 or more than 32.

TwofishEncryptECB

Encrypts input plaintext in the ECB mode.

Syntax

```
IppStatus ippsTwofishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int
    scrlen, const IppsTwofishSpec* pCtx, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srclen`	Length of the input data stream in bytes.
`pCtx`	Pointer to the `IppsTwofishSpec` context.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

`ippStsNoErr`	Indicates no error. Any other value indocates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

ippStsLengthErr	Indicates an error condition if the input data stream length is less than or equal to 0.
ippStsUnderRunErr	Indicates an error condition if *srclen* is not divisible by data block size.
ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptECB

Decrypts byte data stream according to Twofish scheme in the EBC mode.

Syntax

```
IppStatus ippsTwofishDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, const IppsTwofishSpec* pCtx, IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream of variable length.
pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext data stream in bytes.
pCtx	Pointer to the `IppsTwofishSpec` context.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if the output data stream length is less than or equal to zero.

ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.
ippStsUnderRunErr	Indicates an error condition if *srclen* is not divisible by cipher block size.

TwofishEncryptCBC

Encrypts byte data stream according to Twofish scheme in the CBC mode.

Syntax

```
IppStatus ippsTwofishEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    const IppsTwofishSpec* pCtx, const Ipp8u* pIV,  IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the plaintext data stream length in bytes.
pCtx	Pointer to the `IppsTwofishSpec` context.
pIV	Pointer to the initialization vector for the CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if the input data stream length is less than or equal to zero.
ippStsUnderRunErr	Indicates an error condition if *srclen* is not divisible by data block size.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCBC

Decrypts byte data stream according to Twofish scheme in the CBC mode.

Syntax

```
IppStatus ippsTwofishDecryptCBC(const  Ipp8u* pSrc, Ipp8u *pDst, int srclen,
    const IppsTwofishSpec* pCtx, const Ipp8u* pIV,  IppsCPPadding padding);
```

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of the variable length.
srclen	Length of the ciphertext data stream length in bytes.
pCtx	Pointer to the `IppsDESSpec` context.
pIV	Pointer to the initialization vector for CBC mode operation.
padding	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

`ippStsUnderRunErr`	Indicates an error condition if *`srclen`* is not divisible by cipher block size.

TwofishEncryptCFB

Encrypts byte data stream according to Twofish scheme in the CFB mode.

Syntax

```
IppStatus ippsTwofishEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srcLen, int cfbBlkSize, const IppsTwofishSpec* pCtx,  const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srcLen`	Length of the plaintext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsTwofishSpec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.

`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by *`cfbBlkSize`* parameter value.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCFB

Decrypts byte data stream according to Twofish scheme in the CFB mode.

Syntax

```
IppStatus ippsTwofishDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int
    srclen, int cfbBlkSize, const IppsTwofishSpec* pCtx, const Ipp8u*
    pIV, IppsCPPadding padding);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of variable length.
`srclen`	Length of the ciphertext data stream in bytes.
`cfbBlkSize`	Size of the CFB block in bytes.
`pCtx`	Pointer to the `IppsTwofishSpec` context.
`pIV`	Pointer to the initialization vector for the CFB mode operation.
`padding`	`IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCFBSizeErr`	Indicates an error condition if the value for *`cfbBlkSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr`	Indicates an error condition if *`srcLen`* is not divisible by cipher block size.

TwofishEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsTwofishEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsTwofishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input plaintext data stream of a variable length.
`pDst`	Pointer to the resulting ciphertext data stream.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsTwofishSpec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *`crtNumBitSize`* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsTwofishDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,
    const IppsTwofishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

`pSrc`	Pointer to the input ciphertext data stream.
`pDst`	Pointer to the resulting plaintext data stream of a variable length.
`srcLen`	Length of the plaintext data stream in bytes.
`pCtx`	Pointer to the `IppsTwofishSpec` context.
`pCtrValue`	Pointer to the counter data block.
`ctrNumBitSize`	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsCTRSizeErr`	Indicates an error condition if the value of the *crtNumBitSize* is illegal.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Example 2-4 Twofish Encryption and Decryption

```
// use of the FCB mode
void TF_sample(void){
   // size of the Twofish algorithm block is equal to 16
   const int tfBlkSize = 16;

   // get the size of the context needed for the encryption/decryption operation
   int ctxSize;
   ippsTwofishGetSize(&ctxSize);

   // and allocate one
   IppsTwofishSpec* pCtx = (IppsTwofishSpec*)( new Ipp8u [ctxSize] );

   // define the key
   // note that the key length may vary from 16 to 32 bytes
   Ipp8u key[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                  0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15,
                  0x16,0x17};
```

Example 2-4 Twofish Encryption and Decryption (continued)

```
// and prepare the context for Twofish usage
ippsTwofishInit(key,sizeof(key), pCtx);

// define the message to be encrypted
Ipp8u ptext[] = {"quick brown fox jupm over lazy dog"};

// fcb value (bytes)
const int cfbBlkSize = 2;

// define an initial vector
Ipp8u iv[tfBlkSize] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
                       0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// allocate enough memory for the ciphertext
// note that
// the size of the ciphertext is always multiple of the cfbBlkSize size
Ipp8u ctext[(sizeof(ptext)+cfbBlkSize-1) &~(cfbBlkSize-1)];

// encrypt (CFB mode) ptext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be encrypted
ippsTwofishEncryptCFB(ptext, ctext, sizeof(ptext), cfbBlkSize,
                      pCtx,
                      iv,
                      IppsCPPaddingPKCS7);
// allocate memory for the decrypted message
Ipp8u rtext[sizeof(ptext)];
// decrypt (CFB mode) ctext message
```

Example 2-4 Twofish Encryption and Decryption (continued)

```
// pay attention to the 'length' parameter
   // it defines the number of bytes to be decrypted
ippsTwofishDecryptCFB(ctext, rtext, sizeof(ptext), cfbBlkSize,
                         pCtx,
                         iv,
                         IppsCPPaddingPKCS7);
   delete (Ipp8u*)pCtx;
}
```

ARCFour Functions

As the RC4* stream cipher, widely used for file encryption and secure communications, is the property of RSA Security Inc., a cipher discussed in this section and resulting in the same encryption/decryption as RC4* is called ARCFour.

The ARCFour stream cipher ([AC]) uses a variable length key of up to 256 octets (bytes). ARCFour operates in the Output Feedback mode (OFB), defined in [NIST SP 800-38A], which creates the keystream independently of both the plaintext and the ciphertext.

The ARCFour algorithm functions, described in this section, use the context `IppsARCFourState` as an operational vehicle to carry variables needed to execute the algorithm: S-Boxes and a current pair of indices.

The typical application code for conducting an encryption or decryption using ARCFour should follow the sequence of operations listed below:

1. Get the buffer size required to configure the context `IppsARCFourState` by calling the function `ARCFourGetSize`.
2. Call the operating system memory allocation service function to allocate a buffer whose size is not less than the one specified by the function `ARCFourGetSize`. Initialize the pointer *`pCtx`* to the `IppsARCFourState` context by calling the function `ARCFourInit` with the allocated buffer and the respective ARCFour cipher key of the specified size.
3. Call the `ARCFourEncrypt` or `ARCFourDecrypt` function to encrypt or decrypt the input data stream, respectively.
4. Call the operating system memory free service function to release the buffer allocated for the `IppsARCFourState` context, if needed.

ARCFourGetSize

Gets the size of the `IppsARCFourState` *context.*

Syntax

```
IppStatus ippsARCFourGetSize(int* pSize);
```

Parameters

pSize	Pointer to the size value of the `IppsARCFourState` context.

Description

This function is declared in the `ippcp.h` file. The function gets the size of the `IppsARCFourState` context in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if the specified pointer is NULL.

ARCFourCheckKey

Checks weakness of a user-defined key.

Syntax

```
IppStatus ippsARCFourCheckKey(const Ipp8u* pKey, int keyLen, IppsBool*
    pIsWeak);
```

Parameters

pKey	Pointer to the user-defined key.
keyLen	Length of the user-defined key in octets.
pIsWeak	Pointer to the result of checking.

Description

This function is declared in the `ippcp.h` file. The function checks weakness of user-defined key. The function allows to make sure that the supplied key provides sufficient security.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* <1 or *`keyLen`* >256.

ARCFourInit

Initializes user-supplied memory as the `IppsARCFourState` *context for future use.*

Syntax

```
IppStatus ippsARCFourInit(const Ipp8u* pKey, int keyLen,
    IppsARCFourState* pCtx);
```

Parameters

`pKey`	Pointer to the user-defined key.
`keyLen`	Length of the user-defined key in octets.
`pCtx`	Pointer to the `IppsARCFourState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as `IppsARCFourState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* <1 or *`keyLen`* >256.

ARCFourEncrypt

Encrypts a variable length data stream according to ARCFour.

Syntax

```
IppStatus ippsARCFourEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    IppsARCFourState* pCtx);
```

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the plaintext data stream in octets.
pCtx	Pointer to the ARCFourState context.

Description

This function is declared in the ippcp.h file. The function encrypts the input data stream of a variable length using the ARCFour algorithm.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if length of the input data stream is less than one octet.
ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.

ARCFourDecrypt

Decrypts a variable length data stream according to ARCFour.

Syntax

```
IppStatus ippsARCFourDecrypt(const Ipp8u* pSrc, Ipp8u* pDst, int srclen,
    IppsARCFourState* pCtx);
```

Parameters

pSrc	Pointer to the input ciphertext data stream of variable length.
pDst	Pointer to the resulting plaintext data stream.
srclen	Length of the ciphertext data stream in octets.
pCtx	Pointer to the `ARCFourState` context.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ARCFour algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if length of the input data stream is less than one octet.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

ARCFourReset

Resets the `IppsARCFourState` *context to the initial state.*

Syntax

```
IppStatus ippsARCFourReset(IppsARCFourState* pCtx);
```

Parameters

pCtx	Pointer to the `IppsARCFourState` context being reset.

Description

This function is declared in the `ippcp.h` file. The function resets the `IppsARCFourState` context to the state it had immediately after the `ARCFourInit` function call. Contrary to `ARCFourInit`, `ARCFourReset` requires no secret key to initialize the S-Box.

Return Values

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

One-Way Hash Primitives 3

Hash functions are used in cryptography with digital signatures and for ensuring data integrity.

When used with digital signatures, a publicly available hash function hashes the message and signs the resulting hash value. The party who receives the message can then hash the message and check if the block size is authentic for the given hash value.

Hash functions are also referred to as “message digests” and “one-way encryption functions”. Both terms are appropriate since hash algorithms do not have a key like symmetric and asymmetric algorithms and you can recover neither the length nor the contents of the plaintext message from the ciphertext.

To ensure data integrity, hash functions are used to compute the hash value that corresponds to a particular input. Then, if necessary, you can check if the input data has remained unmodified; you can re-compute the hash value again using the available input and compare it to the original hash value.

The Hash Functions section of this chapter describes functions that implement the following hash algorithms for streaming messages: MD5 [RFC 1321], SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 [FIPS PUB 180-2]. These algorithms are widely used in enterprise applications nowadays.

Subsequent sections of the chapter describe Generalized Hash Functions for Non-Streaming Messages, which apply hash algorithms to entire (non-streaming) messages, and Mask Generation Functions, whose algorithms are often based on hash computations.

Each of the above algorithms is implemented as a set of primitive functions.

The full list of Intel® Integrated Performance Primitives (Intel® IPP) Hash Primitive Functions is given in Table 3-1.

Table 3-1 One-way Hash Primitive Functions

Function Base Name	Operation
Hash Functions	
MD5GetSize	Gets the size of the `IppsMD5State` context.
MD5Init	Initializes user supplied memory as `IppsMD5State` context for future use.
MD5Duplicate	Copies one `IppsMD5State` context to another.
MD5Update	Digests the current input message stream of the specified length.
MD5Final	Completes computation of MD5 digest value.
SHA1GetSize	Gets the size of the SHA1 context.
SHA1Init	Initializes user supplied memory as `IppsSHA1State` context for future use.
SHA1Duplicate	Copies one `IppsSHA1State` context to another.
SHA1Update	Digests the current input message stream of the specified length.
SHA1Final	Completes computation of SHA-1 digest value.
SHA224GetSize	Gets the size of the `IppsSHA224State` context.
SHA224Init	Initializes user supplied memory as `IppsSHA224State` context for future use.
SHA224Duplicate	Copies one `IppsSHA224State` context to another.
SHA224Update	Digests the current input message stream of the specified length.
SHA224Final	Completes computation of SHA-224 digest value.
SHA256GetSize	Gets the size of the `IppsSHA256State` context.
SHA256Init	Initializes user supplied memory as `IppsSHA256State` context for future use.
SHA256Duplicate	Copies one `IppsSHA256State` context to another.
SHA256Update	Digests the current input message stream of the specified length.
SHA256Final	Completes computation of SHA-256 digest value.
SHA384GetSize	Gets the size of the `IppsSHA384State` context.
SHA384Init	Initializes user supplied memory as `IppsSHA384State` context for future use.
SHA384Duplicate	Copies one `IppsSHA384State` context to another.
SHA384Update	Digests the current input message stream of the specified length.
SHA384Final	Completes computation of SHA-384 digest value.

Table 3-1 One-way Hash Primitive Functions (continued)

Function Base Name	Operation
SHA512GetSize	Gets the size of the `IppsSHA512State` context.
SHA512Init	Initializes user supplied memory as `IppsSHA512State` context for future use.
SHA512Duplicate	Copies one `IppsSHA512State` context to another.
SHA512Update	Digests the current input message stream of the specified length.
SHA512Final	Completes computation of SHA-512 digest value.
Generalized Hash Functions for Non-Streaming Messages	
MD5MessageDigest	Computes MD5 digest value of the input message.
SHA1MessageDigest	Computes SHA-1 digest value of the input message.
SHA256MessageDigest	Computes SHA-224 digest value of the input message.
SHA256MessageDigest	Computes SHA-256 digest value of the input message.
SHA384MessageDigest	Computes SHA-384 digest value of the input message.
SHA512MessageDigest	Computes SHA-512 digest value of the input message.
Mask Generation Functions	
MGF_MD5	Generates a pseudorandom mask of the specified length using MD5 hash function.
MGF_SHA1	Generates a pseudorandom mask of the specified length using SHA-1 hash function.
MGF_SHA224	Generates a pseudorandom mask of the specified length using SHA-224 hash function.
MGF_SHA256	Generates a pseudorandom mask of the specified length using SHA-256 hash function.
MGF_SHA384	Generates a pseudorandom mask of the specified length using SHA-384 hash function.
MGF_SHA512	Generates a pseudorandom mask of the specified length using SHA-512 hash function.

Hash Functions

Functions featured in this section apply hash algorithms to digesting streaming messages. A primitive implementing a hash algorithm uses the `State` context as an operational vehicle to carry all necessary variables to manage the computation of the chaining digest value. For example, the primitive implementing the SHA-1 hash algorithm must use the `ippsSHA1State` context.

The function `Init` initializes (`MD5Init`, `SHA1Init`, `SHA224Init`, `SHA256Init`, `SHA384Init`, and `SHA512Init`) the context and sets up specified initialization vectors. Once initialized, the function `Update` (`MD5Update`, `SHA1Update`, `SHA224Update`, `SHA256Update`, `SHA384Update`, and `SHA512Update`) digests the input message stream with the selected hash algorithm till it exhausts all message blocks. The function `Final` (`MD5Final`, `SHA1Final`, `SHA224Final`, `SHA256Final`, `SHA384Final`, and `SHA512Final`) is designed to pad the partial message block into a final message block with the specified padding scheme, and then uses the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply the implemented SHA-1 hash standard to digest the input message stream.

1. Call the function `SHA1GetSize` to get the size required to configure the `ippsSHA1State` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `SHA1Init` function to set up the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function `SHA1Update` to digest incoming message stream in the queue till its completion.
4. Call the function `SHA1Final` for padding the partial block into a final SHA-1 message block and transforming it into a 160-bit message digest value.
5. Call the operating system memory free service function to release the `ippsSHA1State` context.

MD5GetSize

Gets the size of the `IppsMD5State` *context in bytes.*

Syntax

```
IppStatus ippsMD5GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsMD5State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsMD5State` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

MD5Init

Initializes user supplied memory as `IppsMD5State` *context for future use.*

Syntax

```
IppStatus ippsMD5Init(IppsMD5State* pCtx);
```

Parameters

pCtx	Pointer to the `IppsMD5State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsMD5State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

MD5Duplicate

Copies one `IppsMD5State` *context to another.*

Syntax

```
IppStatus ippsMD5Duplicate(const IppsMD5State* pSrcCtx, IppsMD5State* pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `IppsMD5State` context.
pDstCtx	Pointer to the `IppsMD5State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsMD5State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

MD5Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsMD5Update(const Ipp8u *pSrcMesg, int mesglen, IppsMD5State
    *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.

mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsMD5State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

MD5Final

Completes computation of the MD5 digest value.

Syntax

```
IppStatus ippsMD5Final(Ipp8u *pMD, IppsMD5State *pCtx);
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsMD5State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA1GetSize

Gets the size of the `IppsSHA1State` *context in bytes.*

Syntax

```
IppStatus ippsSHA1GetSize(int *pSize);
```

Parameters

`pSize`	Pointer to the `IppsSHA1State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA1State` context size in bytes and stores it in *`*pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA1Init

Initializes user supplied memory as `IppsSHA1State` *context for future use.*

Syntax

```
IppStatus ippsSHA1Init(IppsSHA1State* pCtx);
```

Parameters

pCtx	Pointer to the `IppsSHA1State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA1State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA1Duplicate

Copies one `IppsSHA1State` *context to another.*

Syntax

```
IppStatus ippsSHA1Duplicate(const IppsSHA1State* pSrcCtx, IppsSHA1State*
    pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `IppsSHA1State` context.
pDstCtx	Pointer to the `IppsSHA1State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA1State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA1Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA1Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsSHA1State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsSHA1State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA1Final

Completes computation of the SHA1 digest value.

Syntax

```
IppStatus ippsSHA1Final(Ipp8u *pMD, IppsSHA1State *pCtx);
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA1State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA224GetSize

Gets the size of the `IppsSHA224State` *context in bytes.*

Syntax

```
IppStatus ippsSHA224GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsSHA224State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA224State` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA224Init

Initializes user supplied memory as `IppsSHA224State` *context for future use.*

Syntax

```
IppStatus ippsSHA224Init(IppsSHA224State* pCtx);
```

Parameters

pCtx	Pointer to the `IppsSHA224State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA224State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA224Duplicate

Copies one `IppsSHA224State` *context to another.*

Syntax

```
IppStatus ippsSHA224Duplicate(const IppsSHA224State* pSrcCtx,
    IppsSHA224State* pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `SHA224State` context.
pDstCtx	Pointer to the `IppsSHA224State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA224State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA224Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA224Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsSHA224State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsSHA224State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA224Final

Completes computation of the SHA224 digest value.

Syntax

```
IppStatus ippsSHA224Final(Ipp8u *pMD, IppsSHA224State *pCtx);
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA224State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA256GetSize

Gets the size of the `IppsSHA256State` *context in bytes.*

Syntax

```
IppStatus ippsSHA256GetSize(int *pSize);
```

Parameters

`pSize`	Pointer to the `IppsSHA256State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA256State` context size in bytes and stores it in *`*pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA256Init

Initializes user supplied memory as `IppsSHA256State` *context for future use.*

Syntax

```
IppStatus ippsSHA256Init(IppsSHA256State *pCtx);
```

Parameters

pCtx	Pointer to the `IppsSHA256State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA256State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA256Duplicate

Copies one `IppsSHA256State` *context to another.*

Syntax

```
IppStatus ippsSHA256Duplicate(const IppsSHA256State* pSrcCtx,
    IppsSHA256* pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `IppsSHA256State` context.
pDstCtx	Pointer to the `IppsSHA256State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA256State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA256Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA256Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsSHA256State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsSHA256State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA256Final

Completes computation of the SHA256 digest value.

Syntax

```
IppStatus ippsSHA256Final(Ipp8u *pMD, IppsSHA256State *pCtx);
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA256State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA384GetSize

Gets the size of the `IppsSHA384State` *context in bytes.*

Syntax

```
IppStatus ippsSHA384GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsSHA384State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA384State` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA384Init

Initializes user supplied memory as `IppsSHA384State` *context for future use.*

Syntax

```
IppStatus ippsSHA384Init(IppsSHA384State* pCtx);
```

Parameters

pCtx	Pointer to the `IppsSHA384State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA384State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA384Duplicate

Copies one `IppsSHA384State` *context to another.*

Syntax

```
IppStatus ippsSHA384Duplicate(const IppsSHA384State* pSrcCtx,
    IppsSHA384State* pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `IppsSHA384State` context.
pDstCtx	Pointer to the `IppsSHA384State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA384State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA384Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA384Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsSHA384State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsSHA384State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA384Final

Completes computing of the SHA384 digest value.

Syntax

```
IppStatus ippsSHA384Final(Ipp8u *pMD, IppsSHA384State *pCtx );
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA384State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA512GetSize

Gets the size of the `IppsSHA512State` *context in bytes.*

Syntax

```
IppStatus ippsSHA512GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsSHA512State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA512State` context size in bytes and stores it in `*pSize`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA512Init

Initializes user supplied memory as `IppsSHA512State` *context for future use.*

Syntax

```
IppStatus ippsSHA512Init(IppsSHA512State* pState);
```

Parameters

`pCtx`	Pointer to the `IppsSHA512State` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as `IppsSHA512State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SHA512Duplicate

Copies one `IppsSHA512State` *context to another.*

Syntax

```
IppStatus ippsSHA512Duplicate(const IppsSHA512State* pSrcCtx,
    IppsSHA512* pDstCtx)
```

Parameters

pSrcCtx	Pointer to the source `IppsSHA512State` context.
pDstCtx	Pointer to the `IppsSHA512State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA512State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

SHA512Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA512Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsSHA512State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsSHA512State` context.

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA512Final

Completes computation of the SHA512 digest value.

Syntax

```
IppStatus ippsSHA512Final(Ipp8u *pMD, IppsSHA512State *pCtx);
```

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA512State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

Generalized Hash Functions for Non-Streaming Messages

Intel IPP hash functions `MD5MessageDigest`, `SHA1MessageDigest`, `SHA224MessageDigest`, `SHA256MessageDigest`, `SHA384MessageDigest`, and `SHA512MessageDigest` are used to calculate a digest of an entire (non-streaming) input message by applying a selected hash algorithm.

Having the six hash algorithms currently available in the Intel IPP, you may still prefer to use a different implementation of a hash algorithm, based on some other function. In this case you can also use the IPPCP library. To do this, use the definition of a hash function introduced in IPPCP and given in the subsection below. This definition is also applied to a hash function when it is passed as a parameter that some Public Key Cryptography operations optionally use.

General Definition of a Hash Function

Syntax

```
typedef IppStatus(_STDCALL *IppHASH)(const Ipp8u* pMsg, int msgLen,
    Ipp8u* pMD);
```

Parameters

pMsg	Pointer to the input octet string.
msgLen	Length of the input string in octers.
pMD	Pointer to the output message digest.

Description

This declaration is included in the `ippcp.h` file. The function calculates the digest of a non-streaming message using the implemented hash algorithm.

> **NOTE.** Definition of a hash function used in Intel IPP limits length (in octets) of an input message for any specific hash function by the range of the `int` data type, with the upper bound of 2^{32}-1.

MD5MessageDigest

Computes MD5 digest value of the input message.

Syntax

```
IppStatus ippsMD5MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

Example 3-1 MD5 Digest of a Message

```
void MD5_sample(void){
   // define message
   Ipp8u msg[] = "abcdefghijklmnopqrstuvwxyz";

   // once the whole message is placed into memory,
   // one can use the integrated primitive
   Ipp8u digest[16];
   ippsMD5MessageDigest(msg, strlen((char*)msg), digest);
}
```

SHA1MessageDigest

Computes SHA-1 digest value of the input message.

Syntax

```
IppStatus ippsSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesglen,
    Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

Example 3-2 SHA1 Digest of a Message

```
    //    Compute two SHA1 digests of a message:
    //    1-st will correspond of 1/2 message
    //    2-nd will correspond of whole message
void SHA1_sample(void){
   // get size of the SHA1 context
   int ctxSize;
   ippsSHA1GetSize(&ctxSize);

   // allocate the SHA1 context
   IppsSHA1State* pCtx = (IppsSHA1State*)( new Ipp8u [ctxSize] );

   // and initialize the context
   ippsSHA1Init(pCtx);

   // define a message
   Ipp8u msg[] = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";

   int n;

   // update digest using a piece of message
   for(n=0; n<(sizeof(msg)-1)/2; n++)
      ippsSHA1Update(msg+n, 1, pCtx);
// clone the SHA1 context
   IppsSHA1State* pCtx2 = (IppsSHA1State*)( new Ipp8u [ctxSize] );
   ippsSHA1Init(pCtx2);
   ippsSHA1Duplicate(pCtx, pCtx2);
```

Example 3-2 SHA1 Digest of a Message (continued)

```
 // finalize and extract digest of a half message
   Ipp8u digest[20];
   ippsSHA1Final(digest, pCtx);

   // update digest using the SHA1 clone context
   ippsSHA1Update(msg+n, sizeof(msg)-1-n, pCtx2);

   // finalize and extract digest of a whole message
   Ipp8u digest2[20];
   ippsSHA1Final(digest2, pCtx2);

   delete [] (Ipp8u*)pCtx;
   delete [] (Ipp8u*)pCtx2;
}
```

SHA224MessageDigest

Computes SHA-224 digest value of the input message.

Syntax

```
IppStatus ippsSHA224MessageDigest(const Ipp8u *pSrcMsg, int mesglen,
    Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA256MessageDigest

Computes SHA-256 digest value of the input message.

Syntax

```
IppStatus ippsSHA256MessageDigest(const Ipp8u *pSrcMsg, int mesglen,
    Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA384MessageDigest

Computes SHA-384 digest value of the input message.

Syntax

```
IppStatus ippsSHA384MessageDigest(const Ipp8u *pSrcMsg, int mesglen,
    Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

SHA512MessageDigest

Computes SHA-512 digest value of the input message.

Syntax

```
IppStatus ippsSHA512MessageDigest(const Ipp8u *pSrcMsg, int mesglen,
    Ipp8u *pMD);
```

Parameters

pSrcMesg	Pointer to the input message.
mesglen	Message length in octets.
pMD	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

Mask Generation Functions

Public Key Cryptography frequently uses mask generation functions (MGFs) to achieve a particular security goal. For example, MGFs are used both in RSA-OAEP encryption and RSA-SSA signature schemes.

MGF function takes an octet string of a variable length and generates an octet string of a desired length. MGFs are deterministic, which means that the input octet string completely determines the output one. The output of an MGF should be pseudorandom, that is, infeasible to predict. The provable security of such cryptography schemes as RSA-OAEP or RSA-SSA, relies on the random nature of the MGF output. That is why one-way hash functions is one of the well-known ways to implement an MGF. The exact definition of an MGF based on a one-way hash function may be found in [PKCS 1.2.1].

This section describes MGFs based on widely-used MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 hash algorithms as well as the possibility to use a different implementation of MGF.

NOTE. Intel IPP implementation of MGFs limits length (in octets) of an input message for any specific MGF by the range of the `int` data type, with the upper bound of 2^{32}-1.

User's Implementation of a Mask Generation Function

In case you prefer or have to use a different implementation of an MGF you can still use IPPCP. To do this, use the definition of MGF introduced in the IPPCP library and described in this section. The declaration provided below also defines an MGF when it is used as a parameter in some Public Key Cryptography operations.

Syntax

```
typedef IppStatus(_STDCALL *IppMGF)(const Ipp8u* pSeed, int seedLen,
    Ipp8u* pMask, int maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This declaration is included in the `ippcp.h` file. The function generates an octet string of length *maskLen* according to the implemented algorithm, providing pseudorandom output.

MGF_MD5

Generates a pseudorandom mask of the specified length using MD5 hash function.

Syntax

```
IppStatus ippsMGF_MD5(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int
    maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using MD5 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointer is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA1

Generates a pseudorandom mask of the specified length using SHA-1 hash function.

Syntax

```
IppStatus ippsMGF_SHA1(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int
    maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-1 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointer is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA224

Generates a pseudorandom mask of the specified length using SHA-224 hash function.

Syntax

```
IppStatus ippsMGF_SHA224(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask,
    int maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-224 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointer is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA256

Generates a pseudorandom mask of the specified length using SHA-256 hash function.

Syntax

```
IppStatus ippsMGF_SHA256(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask,
    int maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-256 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointer is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA384

Generates a pseudorandom mask of the specified length using SHA-384 hash function.

Syntax

```
IppStatus ippsMGF_SHA384(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask,
    int maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-384 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA512

Generates a pseudorandom mask of the specified length using SHA-512 hash function.

Syntax

```
IppStatus ippsMGF_SHA512(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask,
    int maskLen);
```

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-512 hash algorithm.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pMask* pointer is NULL.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.

Data Authentication Primitive Functions

4

This chapter describes functions for generating message authentication code using the Keyed Hash Functions (HMAC), and the Data Authentication Functions (DAA) schemes.

HMAC is widely used in the applications requiring message authentication and data integrity check. HMAC was initially put forward in [RFC 2401] and adopted by ANSI X9.71 and [FIPS PUB 198]. DAA is widely used in the applications to detect unauthorized modifications, both intentional and accidental, to data. DAA specification can be found in [FIPS PUB 113].

Keyed Hash Functions

Intel IPP HMAC primitive functions described in this chapter use various HMAC schemes based on one-way hash functions described in the One-Way Hash Primitives chapter.

The full list of Intel® Integrated Performance Primitives (Intel® IPP) HMAC functions is given in Table 4-1.

Table 4-1 Intel IPP HMAC Functions

Function Base Name	Operation
Keyed Hash Primitive Functions	
`HMACSHA1GetSize`	Gets the size of the `IppsHMACSHA1State` context.
`HMACSHA1Init`	Initializes user supplied memory as `IppsHMACSHA1State` context for future use.
`HMACSHA1Duplicate`	Copies one `IppsHMACSHA1State` context to another.
`HMACSHA1Update`	Digests the current input message stream of the specified length.
`HMACSHA1Final`	Completes computation of the HMAC value.

Table 4-1 Intel IPP HMAC Functions (continued)

Function Base Name	Operation
HMACSHA1MessageDigest	Computes the HMAC value of the message.
HMACSHA224GetSize	Gets the size of the IppsHMACSHA224State context.
HMACSHA224Init	Initializes user supplied memory as IppsHMACSHA224State context for future use.
HMACSHA224Duplicate	Copies one IppsHMACSHA224State context to another.
HMACSHA224Update	Digests the current input message stream of the specified length.
HMACSHA224Final	Completes computation of the HMAC value.
HMACSHA224MessageDigest	Computes the HMAC value of the message.
HMACSHA256GetSize	Gets the size of the IppsHMACSHA256State context.
HMACSHA256Init	Initializes user supplied memory as IppsHMACSHA256State context for future use.
HMACSHA256Duplicate	Copies one IppsHMACSHA256State context to another.
HMACSHA256Update	Digests the current input message stream of the specified length.
HMACSHA256Final	Completes computation of the HMAC value.
HMACSHA256MessageDigest	Computes the HMAC value of the message.
HMACSHA384GetSize	Gets the size of the IppsHMACSHA384State context.
HMACSHA384Init	Initializes user supplied memory as IppsHMACSHA384State context for future use.
HMACSHA384Duplicate	Copies one IppsHMACSHA384State context to another.
HMACSHA384Update	Digests the current input message stream of the specified length.
HMACSHA384Final	Completes computation of the HMAC value.
HMACSHA384MessageDigest	Computes the HMAC value of the message.
HMACSHA512GetSize	Gets the size of the IppsHMACSHA512State context.
HMACSHA512Init	Initializes user supplied memory as IppsHMACSHA512State context for future use.
HMACSHA512Duplicate	Copies one IppsHMACSHA512State context to another.
HMACSHA512Update	Digests the current input message stream of the specified length.
HMACSHA512Final	Completes computation of the HMAC value.
HMACSHA512MessageDigest	Computes the HMAC value of the message.
HMACMD5GetSize	Gets the size of the IppsHMACMD5State context.
HMACMD5Init	Initializes user supplied memory as IppsHMACMD5State context for future use.

Table 4-1 Intel IPP HMAC Functions (continued)

Function Base Name	Operation
HMACMD5Duplicate	Copies one IppsHMACMD5State context to another.
HMACMD5Update	Digests the current input message stream of the specified length.
HMACMD5Final	Completes computation of the HMAC value.
HMACMD5MessageDigest	Computes the HMAC value of the message.

Each HMAC scheme is implemented as a set of the primitive functions tabled above.

For example, the primitive implementing HMAC which is based on SHA-1 hash algorithm uses the ippsHMACSHA1State context as an operational vehicle to carry all necessary variables to manage computation of chaining digest value.

The function HMACSHA1Init initializes the context and sets up the specified initialization vectors. Once initialized, the function HMACSHA1Update digests the input message stream with the selected hash algorithm till it exhausts all message blocks.

The function HMACSHA1Final is designed to pad the partial message block into a final message block with the specified padding scheme, and then uses the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply the implemented HMAC-SHA1 hash standard to digest the input message stream:

1. Call the function HMACSHA1GetSize to get the size required to configure the IppsHMACSHA1State context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the function HMACSHA1Init to set up key material and the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function HMACSHA1Update to digest incoming message stream in the queue till its completion.
4. Call the function HMACSHA1Final for padding the partial block into a final SHA-1 message block and transforming it into s resultant HMAC value.
5. Call the operating system memory free service function to release IppsHMACSHA1State context.

HMACSHA1GetSize

Gets the size of the `IppsHMACSHA1State` *context.*

Syntax

```
IppStatus ippsHMACSHA1GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsHMACSHA1State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA1State` context size in bytes and stores it in *pSize*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACSHA1Init

Initializes user supplied memory as `IppsHMACSHA1State` *context for future use.*

Syntax

```
IppStatus ippsHMACSHA1Init(const Ipp8u *pKey, int keyLen,
    IppsHMACSHA1State *pCtx;
```

Parameters

pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pCtx	Pointer to the `IppsHMACSHA1State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as the `IppsHMACSHA1State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *`pKey`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* is less than one.

HMACSHA1Duplicate

Copies one `IppsHMACSHA1State` *context to another.*

Syntax

```
IppStatus ippsHMACSHA1Duplicate(const IppsHMACSHA1State* pSrcCtx,
    IppsHMACSHA1State* pDstCtx);
```

Parameters

`pSrcCtx`	Pointer to the source `IppsHMACSHA1State` context.
`pDstCtx`	Pointer to the source `IppsHMACSHA1State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA1State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

HMACSHA1Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACSHA1Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACSHA1State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACSHA1State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACSHA1Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA1Final(Ipp8u *pMAC, int macLen, IppsHMACSHA1State *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.
pCtx	Pointer to the `IppsHMACSHA1State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *macLen* is less than one or greater than the length of the hash value.

HMACSHA1MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesgLen,
    const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMesg	Pointer to the input message.
mesgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero and *macLen* is less than one or greater than the length of the hash value.

HMACSHA224GetSize

Gets the size of the `IppsHMACSHA224State` *context.*

Syntax

```
IppStatus ippsHMACSHA224GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsHMACSHA224State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA224State` context size in bytes and stores it in *`pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACSHA224Init

Initializes user supplied memory as `IppsHMACSHA224State` *context for future use.*

Syntax

```
IppStatus ippsHMACSHA224Init(const Ipp8u *pKey, int keyLen,
    IppsHMACSHA224State *pCtx);
```

Parameters

`pKey`	Pointer to the user supplied key.
`keyLen`	Key length in bytes.
`pCtx`	Pointer to the `IppsHMACSHA224State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as the `IppsHMACSHA224State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *`pKey`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* is less than one.

HMACSHA224Duplicate

Copies one `IppsHMACSHA224State` *context to another.*

Syntax

```
IppStatus ippsHMACSHA224Duplicate(const IppsHMACSHA224State* pSrcCtx,
    IppsHMACSHA224State* pDstCtx);
```

Parameters

pSrcCtx	Pointer to the source `IppsHMACSHA224State` context.
pDstCtx	Pointer to the source `IppsHMACSHA224State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA224State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	ndicates an error condition if the context parameter does not match the operation.

HMACSHA224Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACSHA224Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACSHA224State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACSHA224State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACSHA224Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA224Final(Ipp8u *pMAC, int macLen,
    IppsHMACSHA224State *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

pCtx	Pointer to the `IppsHMACSHA224State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *macLen* is less than one or greater than the length of the hash value.

HMACSHA224MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACSHA224MessageDigest(const Ipp8u *pSrcMesg, int
    mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMesg	Pointer to the input message.
mesgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *`pKey`* of the specified key length *`keyLen`* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *`pMAC`* of the specified length *`macLen`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`msgLen`* is less than zero and *`macLen`* is less than one or greater than the length of the hash value.

HMACSHA256GetSize

Gets the size of the `IppsHMACSHA256State` *context.*

Syntax

```
IppStatus ippsHMACSHA256GetSize(int *pSize);
```

Parameters

`pSize`	Pointer to the `IppsHMACSHA256State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA256State` context size in bytes and stores it in *`pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACSHA256Init

Initializes user supplied memory as `IppsHMACSHA256State` *context for future use.*

Syntax

```
IppStatus ippsHMACSHA256Init(const Ipp8u *pKey, int keyLen,
    IppsHMACSHA256State *pCtx);
```

Parameters

pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pCtx	Pointer to the `IppsHMACSHA256State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsHMACSHA256State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *pKey*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keyLen* is less than one.

HMACSHA256Duplicate

Copies one `IppsHMACSHA256State` *context to another.*

Syntax

```
IppStatus ippsHMACSHA256Duplicate(const IppsHMACSHA256State* pSrcCtx,
    IppsHMACSHA256State* pDstCtx);
```

Parameters

pSrcCtx	Pointer to the source `IppsHMACSHA256State` context.
pDstCtx	Pointer to the source `IppsHMACSHA256State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA256State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	ndicates an error condition if the context parameter does not match the operation.

HMACSHA256Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACSHA256Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACSHA256State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACSHA256State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACSHA256Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA256Final(Ipp8u *pMAC, int macLen,
    IppsHMACSHA256State *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

pCtx	Pointer to the `IppsHMACSHA256State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *macLen* is less than one or greater than the length of the hash value.

HMACSHA256MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACSHA256MessageDigest(const Ipp8u *pSrcMesg, int
    mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMesg	Pointer to the input message.
mesgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero and *macLen* is less than one or greater than the length of the hash value.

Example 4-1 SHA256 HMACs of a Message

```
   // Compute two keyed-hash based message authentication code
   // using the HMAC scheme
   //    1-st will correspond of 1/2 message
   //    2-nd will correspond of whole message

void HMACSHA256_sample(void){
   // define key
   Ipp8u key[] = "the key for HMAC scheme";

   // get size of HMACSHA256 context
   int ctxSize;
   ippsHMACSHA256GetSize(&ctxSize);

   // allocate HMACSHA256 context
   IppsHMACSHA256State* pCtx = (IppsHMACSHA256State*)( new Ipp8u [ctxSize] );

   // and ini context
   ippsHMACSHA256Init(key, strlen((char*)key), pCtx);
```

Example 4-1 SHA256 HMACs of a Message (continued)

```
 // define message
   Ipp8u msg[] = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";

   int n;
  // update MAC using piece of message
   for(n=0; n<(sizeof(msg)-1)/2; n++)
      ippsHMACSHA256Update(msg+n, 1, pCtx);

// clone HMACSHA256 context
   IppsHMACSHA256State* pCtx2 = (IppsHMACSHA256State*)( new Ipp8u [ctxSize] );
   ippsHMACSHA256Init(key, strlen((char*)key), pCtx2);
   ippsHMACSHA256Duplicate(pCtx, pCtx2);

   // finalize and extract digest of a half message
   const int macLen = 16;
   Ipp8u mac[macLen];
   ippsHMACSHA256Final(mac, macLen, pCtx);

   // update MAC using HMACSHA256 clone context
   ippsHMACSHA256Update(msg+n, sizeof(msg)-1-n, pCtx2);

   // finalize and extract digest of a whole message
   Ipp8u mac2[macLen];
   ippsHMACSHA256Final(mac2, macLen, pCtx2);

   delete [] (Ipp8u*)pCtx;
   delete [] (Ipp8u*)pCtx2;
}
```

HMACSHA384GetSize

Gets the size of the `IppsHMACSHA384State` *context.*

Syntax

```
IppStatus ippsHMACSHA384GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsHMACSHA384State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA384State` context size in bytes and stores it in *pSize*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACSHA384Init

Initializes user supplied memory as `IppsHMACSHA384State` *context for future use.*

Syntax

```
IppStatus ippsHMACSHA384Init(const Ipp8u *pKey, int keyLen,
    IppsHMACSHA384State *pCtx);
```

Parameters

pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pCtx	Pointer to the `IppsHMACSHA384State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as the `IppsHMACSHA384State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *`pKey`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* is less than one.

HMACSHA384Duplicate

Copies one `IppsHMACSHA384State` *context to another.*

Syntax

```
IppStatus ippsHMACSHA384Duplicate(const IppsHMACSHA384State* pSrcCtx,
    IppsHMACSHA384State* pDstCtx);
```

Parameters

`pSrcCtx`	Pointer to the source `IppsHMACSHA384State` context.
`pDstCtx`	Pointer to the source `IppsHMACSHA384State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA384State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	ndicates an error condition if the context parameter does not match the operation.

HMACSHA384Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACSHA384Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACSHA384State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACSHA384State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACSHA384Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA384Final(Ipp8u *pMAC, int macLen,
    IppsHMACSHA384State *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.
pCtx	Pointer to the `IppsHMACSHA384State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *macLen* is less than one or greater than the length of the hash value.

HMACSHA384MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACSHA384MessageDigest(const Ipp8u *pSrcMesg, int
    mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMesg	Pointer to the input message.
mesgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero and *macLen* is less than one or greater than the length of the hash value.

HMACSHA512GetSize

Gets the size of the `IppsHMACSHA512State` *context.*

Syntax

```
IppStatus ippsHMACSHA512GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsHMACSHA512State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA512State` context size in bytes and stores it in *`pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACSHA512Init

Initializes user supplied memory as `IppsHMACSHA512State` *context for future use.*

Syntax

```
IppStatus ippsHMACSHA512Init(const Ipp8u *pKey, int keyLen,
    IppsHMACSHA512State *pCtx);
```

Parameters

`pKey`	Pointer to the user supplied key.
`keyLen`	Key length in bytes.
`pCtx`	Pointer to the `IppsHMACSHA512State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as the `IppsHMACSHA512State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *`pKey`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keyLen`* is less than one.

HMACSHA512Duplicate

Copies one `IppsHMACSHA512State` *context to another.*

Syntax

```
IppStatus ippsHMACSHA512Duplicate(const IppsHMACSHA512State* pSrcCtx,
    IppsHMACSHA512State* pDstCtx);
```

Parameters

pSrcCtx	Pointer to the source `IppsHMACSHA512State` context.
pDstCtx	Pointer to the source `IppsHMACSHA512State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA512State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	ndicates an error condition if the context parameter does not match the operation.

HMACSHA512Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACSHA512Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACSHA512State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACSHA512State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACSHA512Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA512Final(Ipp8u *pMAC, int macLen,
    IppsHMACSHA512State *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

pCtx	Pointer to the `IppsHMACSHA512State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *macLen* is less than one or greater than the length of the hash value.

HMACSHA512MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACSHA512MessageDigest(const Ipp8u *pSrcMesg, int
    mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMesg	Pointer to the input message.
mesgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *`pKey`* of the specified key length *`keyLen`* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *`pMAC`* of the specified length *`macLen`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`msgLen`* is less than zero and *`macLen`* is less than one or greater than the length of the hash value.

HMACMD5GetSize

Gets the size of the `IppsHMACMD5State` *context.*

Syntax

```
IppStatus ippsHMACMD5GetSize(int *pSize);
```

Parameters

`pSize`	Pointer to the `IppsHMACMD5State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACMD5State` context size in bytes and stores it in *`pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

HMACMD5Init

Initializes user supplied memory as `IppsHMACMD5State` *context for future use.*

Syntax

```
IppStatus ippsHMACMD5Init(const Ipp8u *pKey, int keyLen,
    IppsHMACMD5State *pCtx);
```

Parameters

pKey	Pointer to the user supplied key.
keyLen	Key length in bytes.
pCtx	Pointer to the `IppsHMACMD5State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsHMACMD5State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *pKey*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keyLen* is less than one.

HMACMD5Duplicate

Copies one `IppsHMACMD5State` *context to another.*

Syntax

```
IppStatus ippsHMACMD5Duplicate(const IppsHMACMD5State* pSrcCtx,
    IppsHMACMD5State* pDstCtx);
```

Parameters

pSrcCtx	Pointer to the source `IppsHMACMD5State` context.
pDstCtx	Pointer to the source `IppsHMACMD5State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACMD5State` context to another.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	ndicates an error condition if the context parameter does not match the operation.

HMACMD5Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACMD5Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsHMACMD5State *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part of the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHMACMD5State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

HMACMD5Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACMD5Final(Ipp8u *pMAC, int macLen, IppsHMACMD5State
    *pCtx);
```

Parameters

pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.
pCtx	Pointer to the `IppsHMACMD5State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *`macLen`* is less than one or greater than the length of the hash value.

HMACMD5MessageDigest

Computes the HMAC value of the message.

Syntax

```
IppStatus ippsHMACMD5MessageDigest(const Ipp8u *pSrcMesg, int mesgLen,
    const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

`pSrcMesg`	Pointer to the input message.
`mesgLen`	Message length in bytes.
`pKey`	Pointer to the user supplied key.
`keyLen`	Key length in bytes.
`pMAC`	Pointer to the resultant HMAC value.
`macLen`	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *`pKey`* of the specified key length *`keyLen`* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *`pMAC`* of the specified length *`macLen`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`msgLen`* is less than zero and *`macLen`* is less than one or greater than the length of the hash value.

Example 4-2 MD5 HMAC of a Message

```
void HMACMD5_sample(void){
   // define key
   Ipp8u key[] = "the key for HMAC scheme";

   // define message
   Ipp8u msg[] = "abcdefghijklmnopqrstuvwxyz";

   // as soon as whole message placed into memory
   // one can use integrated primitive
   int macLen = 12;
   Ipp8u mac[16];
   ippsHMACMD5MessageDigest(msg, strlen((char*)msg),
                            key, strlen((char*)key),
                            mac, macLen);
}
```

Data Authentication Functions

Intel IPP Data Authentication (DAA) primitive functions are designed for applications to use various DAA schemes based on the set of symmetric cryptography functions described in the Symmetric Cryptography Primitive Functions chapter.

The implementation of each DAA scheme is presented as a set of primitive functions.

The full list of Intel IPP DAA Functions is given is Table 4-2.

Table 4-2 Intel IPP Data Authentication Functions

Function Base Name	Operation
`DAADESGetSize`	Gets the size of the `IppsDAADESState` context.
`DAADESInit`	Initializes user supplied memory as `IppsDAADESState` context for future use.
`DAADESUpdate`	Digests the current input message stream of the specified length.
`DAADESFinal`	Completes computation of the DAC value.
`DAADESMessageDigest`	Computes the DAC value of the message.
`DAATDESGetSize`	Gets the size of `IppsDAATDESState` context.
`DAATDESInit`	Initializes user supplied memory as `IppsDAATDESState` context for future use.
`DAATDESUpdate`	Digests the current input message stream of the specified length.
`DAATDESFinal`	Completes computation of the DAC value.
`DAATDESMessageDigest`	Computes the DAC value of the message.
`DAARijndael128GetSize`	Gets the size of the `IppsDAARijndael128State` context.
`DAARijndael128Init`	Initializes user supplied memory as `IppsDAARijndael128State` context for future use.
`DAARijndael128Update`	Digests the current input message stream of the specified length.
`DAARijndael128Final`	Completes computation of the DAC value.
`DAARijndael128MessageDigest`	Computes the DAC value of the message.
`DAARijndael192GetSize`	Gets the size of the `IppsDAARijndael192State` context.
`DAARijndael192Init`	Initializes user supplied memory as `IppsDAARijndael192State` context for future use.
`DAARijndael192Update`	Digests the current input message stream of the specified length.
`DAARijndael192Final`	Completes computation of the DAC value.
`DAARijndael192MessageDigest`	Computes the DAC value of the message.

Table 4-2 Intel IPP Data Authentication Functions (continued)

Function Base Name	Operation
DAARijndael256GetSize	Gets the size of the IppsDAARijndael256State context.
DAARijndael256Init	Initializes user supplied memory as IppsDAARijndael256State context for future use.
DAARijndael256Update	Digests the current input message stream of the specified length.
DAARijndael256Final	Completes computation of the DAC value.
DAARijndael256MessageDigest	Computes the DAC value of the message.
DAABlowfishGetSize	Gets the size of the IppsDAABlowfishState context.
DAABlowfishInit	Initializes user supplied memory as IppsDAABlowfishState context for future use.
DAABlowfishUpdate	Digests the current input message stream of the specified length.
DAABlowfishFinal	Completes computation of the DAC value.
DAABlowfishMessageDigest	Computes the DAC value of the message.
DAATwofishGetSize	Gets the size of the IppsDAATwofishState context.
DAATwofishInit	Initializes user supplied memory as IppsDAATwofishState context for future use.
DAATwofishUpdate	Digests the current input message stream of the specified length.
DAATwofishFinal	Completes computation of the DAC value.
DAATwofishMessageDigest	Computes the DAC value of the message.

The primitive implementing a DAA scheme uses the context as the operational vehicle to carry all necessary variables to manage computation of the chaining digest value. For example, the primitive implementing the DAA scheme based on the Rijndael128 block cipher uses `ippsDAARijndael128` context.

The function `Init` (`DAARijndael128Init`, `DAADESInit`, and others) initializes the context and sets up the specified initialization vectors. Once initialized, the function `Update` (`DAARijndael128Update`, `DAADESUpdate`, and others) digests the input message stream with the selected hash algorithm till it exhausts all message blocks. The function `Final` (`DAARijndael128Final`, `DAADESFinal`, and others) is designed to pad the partial message block into a final message block with the specified padding scheme, and further uses the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply an implemented DAA based on the Rijndael128 block cipher to digest the input message stream:

1. Call the function `DAARijndael128GetSize` to get the size required to configure the `ippsDAARijndael128State` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the function `DAARijndael128Init` to set up the initial context state.
3. Keep calling the function `DAARijndael128Update` to digest incoming message stream in the queue till its completion.
4. Call the function `DAARijndael128Final` for padding the final partial block applying Rijndael128 block cipher and transforming ciphertext block into resultant DAA value.
5. Call the operating system memory free service function to release the `IppsDAARijndaelState` context.

DAADESGetSize

Gets the size of the `IppsDAADESState` *context.*

Syntax

```
IppStatus ippsDAADESGetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAADESState` context.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.

DAADESInit

Initializes user supplied memory as `IppsDAADESState` *context for future use.*

Syntax

```
IppStatus ippsDAADESInit(const Ipp8u *pKey, IppsDAADESState *pCtx);
```

Parameters

pKey	Pointer to the DES key.
pCtx	Pointer to the `IppsDAADESState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAADESState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operation.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAADESUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAADESUpdate(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAADESState* pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsDAADESState` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by

applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAADESFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAADESFinal(Ipp8u *pDAC, int dacLen, IppsDAADESState* pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAADESState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if *`dacLen`* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAADESMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAADESMessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, Ipp8u *pMAC, int macLen);
```

Parameters

`pSrcMsg`	Pointer to the input message.
`msgLen`	Message length in bytes.
`pKey`	Pointer to the user supplied key.
`pMAC`	Pointer to the resultant HMAC value.
`macLen`	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *`pKey`* of the specified key length *`keyLen`* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *`pMAC`* of the specified length *`macLen`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`msgLen`* is less than zero, *`macLen`* is less than 1 or greater than cipher's data block length, *`keyLen`* value is illegal.

DAATDESGetSize

Gets the size of the `IppsDAATDESState` *context.*

Syntax

```
IppStatus ippsDAATDESGetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAATDESState` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAATDESInit

Initializes user supplied memory as the `IppsDAATDESState` *context for future use.*

Syntax

```
IppStatus ippsDAATDESInit(const Ipp8u* pKey, const Ipp8u* pKey2, const
    Ipp8u* pKey3, IppsDAATDESState* pCtx);
```

Parameters

pKey	Pointer to the TDES key.
pKey2	Pointer to the TDES key.
pKey3	Pointer to the TDES key.
pCtx	Pointer to the `IppsDAATDESState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAATDESState` context. In addition, `DAATDESInit` uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAATDESUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAATDESUpdate(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAATDESState *pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsDAATDESState` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAATDESFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAATDESFinal(Ipp8u *pDAC, int dacLen, IppsDAATDESState* pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAATDESState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAATDESMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAATDESMessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey1, const Ipp8u *pKey2, const Ipp8u *pKey3, Ipp8u
    *pMAC, int macLen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey1	Pointer to the user supplied key.
pKey2	Pointer to the user supplied key.
pKey3	Pointer to the user supplied key.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey1*, *pKey2*, and *pKey3* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAARijndael128GetSize

Gets the size of the `IppsDAARijndael128State` *context.*

Syntax

```
IppStatus ippsDAARijndael128GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAARijndael128State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAARijndael128Init

Initializes user supplied memory as `IppsDAARijndael128State` *context for future use.*

Syntax

```
IppStatus ippsDAARijndael128Init(const Ipp8u* pKey,
    IppsRijndaelKeyLength keyLen, IppsDAARijndael128State* pCtx);
```

Parameters

pKey	Pointer to the Rijndael128 key.
keyLen	Key byte stream length in bytes defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsDAARijndael128State` being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAARijndael128State` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keyLen* is not equal to `IppsRijndaelKey128`, `IppsRijndaelKey192`, or `IppsRijndaelKey256`.

DAARijndael128Update

Digest the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAARijndael128Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAARijndael128State* pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsDAARijndael128State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael128Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAARijndael128Final(Ipp8u *pDAC, int dacLen,
    IppsDAARijndael128State *pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAARijndael128State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael128MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAARijndael128MessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAARijndael192GetSize

Gets the size of the `IppsDAARijndael192State` *context.*

Syntax

```
IppStatus ippsDAARijndael192GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAARijndael192State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAARijndael192Init

Initializes user supplied memory as `IppsDAARijndael192State` *context for future use.*

Syntax

```
IppStatus ippsRijndael192Init(const Ipp8u* pKey,
    IppsDAARijndaelKeyLength keyLen, IppsRijndael192State* pCtx);
```

Parameters

pKey	Pointer to the Rijndael192 key.
keyLen	Key byte stream length in bytes defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsDAARijndael192State` being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAARijndael192State` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keyLen* is not equal to `IppsRijndaelKey128`, `IppsRijndaelKey192`, or `IppsRijndaelKey256`.

DAARijndael192Update

Digest the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAARijndael192Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAARijndael192State* pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsDAARijndael192State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael192Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAARijndael192Final(Ipp8u *pDAC, int dacLen,
    IppsDAARijndael192State *pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAARijndael192State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael192MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAARijndael192MessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pDAC, int macLen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAARijndael256GetSize

Gets the size of the `IppsDAARijndael256State` *context.*

Syntax

```
IppStatus ippsDAARijndael256GetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAARijndael256State` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAARijndael256Init

Initializes user supplied memory as `IppsDAARijndael256State` *context for future use.*

Syntax

```
IppStatus ippsRijndael256Init(const Ipp8u* pKey,
    IppsDAARijndaelKeyLength keyLen, IppsRijndael256State* pCtx);
```

Parameters

pKey	Pointer to the Rijndael256 key.
keyLen	Key byte stream length in bytes defined by the `IppsRijndaelKeyLength` enumerator.
pCtx	Pointer to the `IppsDAARijndael256State` being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAARijndael256State` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *keyLen* is not equal to `IppsRijndaelKey128`, `IppsRijndaelKey192`, or `IppsRijndaelKey256`.

DAARijndael256Update

Digest the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAARijndael256Update(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAARijndael256State* pCtx);
```

Parameters

pSrcMesg	Pointer to the buffer containing a part or the whole message.
mesglen	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsDAARijndael256State` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael256Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAARijndael256Final(Ipp8u *pDAC, int dacLen,
    IppsDAARijndael256State* pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAARijndael256State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAARijndael256MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAARijndael256MessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pMAC, int maclen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAABlowfishGetSize

Gets the size of the `IppsDAABlowfishState` *context.*

Syntax

```
IppStatus ippsDAABlowfishGetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAABlowfishState` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAABlowfishInit

Initializes user supplied memory as `IppsDAABlowfishState` *context for future use.*

Syntax

```
IppStatus ippsDAABlowfishInit(const Ipp8u* pKey, int keylen,
    IppsDAABlowfishState* pCtx);
```

Parameters

pKey	Pointer to the DAABlowfish key.
keylen	Key byte stream length in bytes.
pCtx	Pointer to the `IppsDAABlowfishState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAABlowfishState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keylen`* is less than 1 or greater than 56.

DAABlowfishUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAABlowfishUpdate(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAABlowfishState* pCtx);
```

Parameters

`pSrcMesg`	Pointer to the buffer containing a part or the whole message.
`mesglen`	Length of the actual part of the message in bytes.
`pCtx`	Pointer to the `IppsDAABlowfishState` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAABlowfishFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAABlowfishFinal(Ipp8u *pDAC, int dacLen,
    IppsDAABlowfishState* pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAABlowfishState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher’s data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAABlowfishMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAABlowfishMessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAATwofishGetSize

Gets the size of the `IppsDAATwofishState` *context.*

Syntax

```
IppStatus ippsDAATwofishGetSize(int *pSize);
```

Parameters

pSize	Pointer to the `IppsDAATwofishState` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

DAATwofishInit

Initializes user supplied memory as `IppsDAATwofishState` *context for future use.*

Syntax

```
IppStatus ippsDAATwofishInit(const Ipp8u* pKey, int keylen,
    IppsDAATwofishState* pCtx);
```

Parameters

pKey	Pointer to the DAATwofish key.
keylen	Key byte stream length in bytes.
pCtx	Pointer to the `IppsDAATwofishState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAATwofishState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`keylen`* is less than 1 or greater than 32.

DAATwofishUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsDAATwofishUpdate(const Ipp8u *pSrcMesg, int mesglen,
    IppsDAATwofishState* pCtx);
```

Parameters

`pSrcMesg`	Pointer to the buffer containing a part or the whole message.
`mesglen`	Length of the actual part of the message in bytes.
`pCtx`	Pointer to the `IppsDAATwofishState` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if the input data stream length is less than zero.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAATwofishFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippsDAATwofishFinal(Ipp8u *pDAC, int dacLen,
    IppsDAATwofishState* pCtx);
```

Parameters

pDAC	Pointer to the DAC value.
dacLen	Specified length of the DAC.
pCtx	Pointer to the `IppsDAATwofishState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *dacLen* is less than 1 or greater than cipher's data block length.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

DAATwofishMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippsDAATwofishMessageDigest(const Ipp8u *pSrcMsg, int msgLen,
    const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int maclen);
```

Parameters

pSrcMsg	Pointer to the input message.
msgLen	Message length in bytes.
pKey	Pointer to the user supplied key.
keyLen	Key length.
pMAC	Pointer to the resultant HMAC value.
macLen	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsLengthErr	Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

Public Key Cryptography Functions

5

This chapter introduces Intel® Integrated Performance Primitives (Intel® IPP) Big Number Arithmetic, Montgomery Reduction Scheme Functions, Pseudorandom Number Generation Functions, Prime Number Generation Functions, RSA Algorithm Functions, Discrete-Logarithm-Based Cryptography Functions, and Elliptic Curve Cryptography Functions (ECC) functions.

Big Number Arithmetic

This section describes primitives for performing arithmetic operations with integer big numbers of variable length.

The full list of functions for big number arithmetic is given in Table 5-1.

Table 5-1 Intel IPP Big Number Arithmetic Functions

Function Base Name	Operation
Unsigned Big Number Arithmetic Functions	
`Add_BNU`	Adds two unsigned integer big numbers of the same length.
`Sub_BNU`	Subtracts one integer big number from another integer big number of the same length.
`MulOne_BNU`	Multiplies unsigned integer big number by 32-bit unsigned integer.
`MACOne_BNU_I`	Computes multiplication of unsigned integer big number by 32-bit integer and accumulates the result with another integer big number.
`Mul_BNU4`	Multiplies two unsigned integers of 4*32 bits.
`Mul_BNU8`	Multiplies two unsigned integers of 8*32 bits.
`Div_64u32u`	Divides unsigned 64-bit integer by unsigned 32-bit integer.

Table 5-1 Intel IPP Big Number Arithmetic Functions (continued)

Function Base Name	Operation
`Sqr_32u64u`	Computes the square of 32-bit words in the input array.
`Sqr_BNU4`	Computes the square of an unsigned integer big number of 4*32 bits.
`Sqr_BNU8`	Computes the square of an unsigned integer big number of 8*32 bits.
`SetOctString_BNU`	Converts octet string into unsigned integer big number.
`GetOctString_BNU`	Converts unsigned integer big number into octet string.
Signed Big Number Arithmetic Functions	
`BigNumGetSize`	Gets the size of the `IppsBigNumState` context.
`BigNumInit`	Initializes context and partitions allocated buffer.
`Set_BN`	Defines the sign and value of the context.
`SetOctString_BN`	Converts octet string into a positive Big Number.
`GetSize_BN`	Returns the maximum length of the integer big number the structure can store.
`Get_BN`	Extracts the sign and value of the integer big number from the input structure.
`GetOctString_BN`	Converts a positive Big Number into octet String.
`Cmp_BN`	Compares two Big Numbers.
`CmpZero_BN`	Checks the value of the input data field.
`Add_BN`	Adds two integer big numbers.
`Sub_BN`	Subtracts one integer big number from another.
`Mul_BN`	Multiplies two integer big numbers.
`MAC_BN_I`	Multiplies two integer big numbers and accumulates the result with the third integer big number.
`Div_BN`	Divides one integer big number by another.
`Mod_BN`	Computes modular reduction for input integer big number with respect to specified modulus.
`Gcd_BN`	Computes the greatest common divisor.
`ModInv_BN`	Computes multiplicative inverse of a positive integer big number with respect to specified modulus.

The magnitude of an integer big number is specified by an array of unsigned integer data type `Ipp32u` *`rp`*`[length]` and corresponds to the mathematical value

$$r = \sum_{0 \le i < length} rp[i] \times 2^{32i}.$$

This section uses the following definition to define the sign of an integer big number:

```
typedef enum {
IppsBigNumNEG=0,
IppsBigNumPOS=1
} IppsBigNumSGN;
```

The functions described in this section use the context `IppsBigNumState` to serve as an operational vehicle that carries not only the sign and value of the data, but also a sufficient working buffer reserved for various arithmetic operations. The length of the context `IppsBigNumState` is defined as the length of the data carried by the structure and the size of the context `IppsBigNumState` is therefore defined as the maximal length of the data that this operational vehicle can carry.

NOTE. In all unsigned big number arithmetic functions described below, integers pointed to by *a*, *b*, and *r* are all of (*n**32) bits.

Add_BNU

Adds two unsigned integer big numbers of the same length.

Syntax

```
IppStatus ippsAdd_BNU(const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n,
    Ipp32u *carry);
```

Parameters

a	First unsigned integer big number of `n*32` bits.
b	Second unsigned integer big number of `n*32` bits.
n	Size specified for the input parameters *a* and *b* and the result parameter *r*. The size is expressed in the number of 32-bit words.

r	On output, addition result.
carry	On output, addition carry. The possible value is 0 or 1.

Description

This function is declared in the `ippcp.h` file. The function adds two unsigned integer big numbers of the same length and returns the result of the operation with a possible carry.

The following pseudocode represents this function:

$(carry|({}^{*}r)) \leftarrow ({}^{*}a) + ({}^{*}b).$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *n* is less than or equal to 0.

Sub_BNU

Subtracts one integer big number from another integer big number of the same length.

Syntax

```
IppStatus ippsSub_BNU(const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n,
    Ipp32u *carry);
```

Parameters

a	First unsigned integer big number of `n*32` bits.
b	Second unsigned integer big number of `n*32` bits.
n	Size specified for the input parameters *a* and *b* and the result parameter *r*. The size is expressed in the number of 32-bit words.
r	Subtraction result.
carry	Subtraction borrow. Possible value is 0 or 1.

Description

This function is declared in the `ippcp.h` file. The function subtracts one integer big number from another big-number integer of the same length and returns the result of the operation with a possible borrow returned as the *carry* argument.

The following pseudo code represents this function:

$(^*r) \leftarrow (^*a) - (^*b)$;

$carry = ((^*a) < (^*b))$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *n* is less than or equal to 0.

MulOne_BNU

Multiplies unsigned integer big number by 32-bit unsigned integer.

Syntax

```
IppStatus ippsMulOne_BNU(const Ipp32u *a, Ipp32u *r, int n, Ipp32u w,
    Ipp32u *carry);
```

Parameters

a	Unsigned integer big number of `n*32` bits.
w	Unsigned long integer of 32 bits serving as multiplier for the operation.
n	Size specified for the input parameter *a* and the result parameter *r*. The size is expressed in the number of 32-bit words.
r	Multiplication result.
carry	Multiplication carry.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplication of an unsigned integer big number *a* by a 32-bit unsigned integer *w*. The function returns the result to the length array *r*, and the carry of the result to *carry*.

The following pseudo code represents this function:

$(carry|(^{*}r)) \leftarrow (^{*}a)w$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *n* is less than or equal to 0.

MACOne_BNU_I

Computes multiplication of unsigned integer big number by 32-bit integer and accumulates the result with another integer big number.

Syntax

```
IppStatus ippsMACOne_BNU_I(const Ipp32u *a, Ipp32u *r, int n, Ipp32u w,
    Ipp32u *carry);
```

Parameters

a	Multiplicand, an unsigned integer big number.
w	32-bit unsigned long integer multiplier.
n	Size specified for the input parameters *a* and the result parameter *r*. The size is expressed in the number of 32-bit words.
r	Unsigned integer big number accumulator.
carry	Operation carry.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplication of an unsigned integer big number *a* by a 32-bit integer *w* and accumulates the result with another integer big number *r* of same length. The result of the operation is returned to *r*, and the carry of the result is returned as *carry*.

The following pseudo code represents this function:

$(carry|(^*r)) \leftarrow (^*r) + (^*a)w.$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *n* is less than or equal to 0.

Mul_BNU4

*Multiplies two unsigned integers of 4*32 bits.*

Syntax

```
IppStatus ippsMul_BNU4(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Parameters

a	Multiplicand, an integer big number of `4*32` bits.
w	Multiplier, an integer big number of `4*32` bits.
r	Multiplication result of `8*32` bits.

Description

This function is declared in the `ippcp.h` file. The function multiplies two unsigned integers of `4*32` bit and returns the result of `8*32` bits.

The following pseudo code represents this function:

$(^*r) \leftarrow (^*a)(^*b).$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Mul_BNU8

*Multiplies two unsigned integers of 8*32 bits.*

Syntax

```
IppStatus ippsMul_BNU8(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Parameters

a	Multiplicand, an integer big number of `8*32` bits.
b	Multiplier, an integer big number of `8*32` bits.
r	Multiplication result of `16*32` bits.

Description

This function is declared in the `ippcp.h` file. The function multiplies two unsigned integers of `8*32` bit and returns the result of `16*32` bits.

The following pseudo code represents this function:

$(^*r) \leftarrow (^*a)(^*b)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Div_64u32u

Divides unsigned 64-bit integer by unsigned 32-bit integer.

Syntax

```
IppStatus ippsDiv_64u32u(Ipp64u a, Ipp32u b, Ipp32u *q, Ipp32u *r);
```

Parameters

a	Dividend of 64 bits.
b	Divisor of 32 bits.
q	Quotient of 32 bits.
r	Remainder of 32 bits.

Description

This function is declared in the `ippcp.h` file. The function divides a 64-bit unsigned integer dividend by a 32-bit unsigned divisor and returns the quotient and remainder.

The following pseudo code represents this function:

$a = b * q + r$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if the quotient is not 32 bits. In this case the result is undefined.
`ippStsDivByZeroErr`	Indicates an error condition if zero divisor is used.

Sqr_32u64u

Computes the square of 32-bit words in the input array.

Syntax

```
IppStatus ippsSqr_32u64u(const Ipp32u *src, int n, Ipp64u *dst);
```

Parameters

src	Array of 32-bit words.
n	Input array size.
dst	Pointer to the result.

Description

This function is declared in the `ippcp.h` file. The function computes the square of each unsigned 32-bit long word in the input array. The result of the operation is stored in the array of 64-bit unsigned integers.

The following pseudo code represents this function:

$dst[i] \leftarrow src[i]*src[i]$ where $i = 0, 1, 2, ..., n-1$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *n* is less than or equal to 0.

Sqr_BNU4

*Computes the square of an unsigned integer big number of 4*32 bits.*

Syntax

```
IppStatus ippsSqr_BNU4(const Ipp32u *a, Ipp32u *r);
```

Parameters

a	Multiplicand of `4*32` bits.
r	Square operation result of `8*32` bits.

Description

This function is declared in the `ippcp.h` file. The function computes the square of an unsigned integer big number of `4*32` bits and stores the result in the memory.

The following pseudo code represents this function:

$(^{*}r) \leftarrow (^{*}a)(^{*}a)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Sqr_BNU8

*Computes the square of an unsigned integer big number of 8*32 bits.*

Syntax

```
IppStatus ippsSqr_BNU8(const Ipp32u *a, Ipp32u *r);
```

Parameters

a	Multiplicand of `8*32` bits.
r	Square operation result of `16*32` bits.

Description

This function is declared in the `ippcp.h` file. The function computes the square of an unsigned integer big number of `8*32` bits and stores the result in the memory.

The following pseudo code represents this function:

```
(*r) <- (*a) (*a)
```

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

SetOctString_BNU

Converts octet string into unsigned integer big number.

Syntax

```
IppStatus ippsSetOctString_BNU(const Ipp8u* pOctStr, int strLen, Ipp32u*
    pBNU, int* pBNUsize);
```

Parameters

pOctStr	Pointer to the input octet string `OctStr`.
strLen	Length of the octet string.
pBNU	Pointer to the unsigned integer big number BNU.
pBNUsize	Pointer to the size (in 32-bit items) of the unsigned integer big number.

Description

The function is declared in the `ippcp.h` file. This function converts octet string into unsigned integer big number.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if specified *strLen* is less than 1.
`ippStsSizeErr`	Indicates an error condition if specified **pBNUsize* is not sufficient for keeping actual *strLen*.

GetOctString_BNU

Converts unsigned integer big number into octet string.

Syntax

```
IppStatus ippsGetOctString_BNU(const Ipp32u* pBNU, int bnuSize, Ipp8u*
    pOctStr, int strLen);
```

Parameters

pBNU	Pointer to the source unsigned integer big number BNU.
bnuSize	Unsigned integer big number size (in 32-bit intems)
pOctStr	Pointer to the source octet string `OctStr`.
strLen	Length of the octet string.

Description

The function is declared in the `ippcp.h` file. This function converts unsigned integer big number BNU into octet string `OctStr`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if specified *bnuSize* is less than 1.
`ippStsRangeErr`	Indicates an error condition if 256^*strLen* is less than the unsigned integer big number.

BigNumGetSize

Gets the size of the `IppsBigNumState` *context in bytes.*

Syntax

```
IppStatus ippsBigNumGetSize(int length, int *size);
```

Parameters

length	Integer big number length in `Ipp32u`.
size	Size of the buffer in bytes required for initialization.

Description

This function is declared in the `ippcp.h` file. The function specifies the buffer size required to define a structuralized working buffer of the context `IppsBigNumState` for the storage and operations on an integer big number in bytes.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.

BigNumInit

Initializes context and partitions allocated buffer.

Syntax

```
IppStatus ippsBigNumInit(int length, IppsBigNumState *b);
```

Parameters

length	Size of the big number for the context initialization.

b	Pointer to the supplied buffer used to store the initialized context `IppsBigNumState`.

Description

This function is declared in the `ippcp.h` file. The function initializes the context `IppsBigNumState` using the specified buffer space and partitions the given buffer to store and execute arithmetic operations on an integer big number of the *`length`* size.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`length`* is less than or equal to 0.

Set_BN

Defines the sign and value of the context.

Syntax

```
IppStatus ippsSet_BN(IppsBigNumSGN sgn, int length, const Ipp32u *data,
    IppsBigNumState *x);
```

Parameters

sgn	Sign of `IppsBigNumState *x`.
length	Array length of the input data.
data	Data array.
x	On output, the context `IppsBigNumState` updated with the input data.

Description

This function is declared in the `ippcp.h` file. The function defines the sign and value for `IppsBigNumState *x` with the specified inputs `IppsBigNumSGN sgn` and `const Ipp32u *data`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.
`ippStsOutOfRangeErr`	Indicates an error condition if *length* is more than the size of `IppsBigNumState *x`.
`ippStsBadArgErr`	Indicates an error condition if the big number is set to zero with the negative sign.

Example 5-1 Create a Big Number

```
IppsBigNumState* New_BN(int size, const Ipp32u* pData=0){
   // get the size of the Big Number context
   int ctxSize;
   ippsBigNumGetSize(size, &ctxSize);
   // allocate the Big Number context
   IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [ctxSize] );
   // and initialize one
   ippsBigNumInit(size, pBN);

   // if any data was supplied, then set up the Big Number value
   if(pData)
      ippsSet_BN(IppsBigNumPOS, size, pData, pBN);

   // return pointer to the Big Number context for future use
   return pBN;
}
```

SetOctString_BN

Converts octet string into a positive Big Number.

Syntax

```
IppStatus ippsSetOctString_BN(const Ipp8u* pOctStr, int strLen,
    IppsBigNumState* pBN);
```

Parameters

pOctStr	Pointer to the input octet string.
strLen	Octet string length in bytes.
pBN	Pointer to the context of the output Big Number.

Description

The function is declared in the `ippcp.h` file. This function converts octet string into a positive Big Number.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if specified *strLen* is less than 1.
`ippStsSizeErr`	Indicates an error condition if insufficient space has been reserved for Big Number.

Example 5-2 Create a Big Number from a String

```
void Set_BN_sample(void){
   // desired value of Big Number is 0x123456789abcdef0fedcba9876543210
   Ipp8u desiredBNvalue[] = "\x12\x34\x56\x78\x9a\xbc\xde\xf0"
                            "\xfe\xdc\xba\x98\x76\x54\x32\x10";

   // estimate required size of Big Number
   //int size = (sizeof(desiredBNvalue)+3)/4;
   int size = (sizeof(desiredBNvalue)-1+3)/4;

   // and create new (and empty) one
   IppsBigNumState* pBN = New_BN(size);

   // set up the value from the srting
   ippsSetOctString_BN(desiredBNvalue, sizeof(desiredBNvalue)-1, pBN);

   Type_BN("Big Number value is:\n", pBN);
}
```

GetSize_BN

Returns the maximum length of the integer big number the structure can store.

Syntax

```
IppStatus ippsGetSize_BN(const IppsBigNumState *b, int *size);
```

Parameters

b	Integer big number of the data type `IppsBigNumState`.
size	Maximum length of the integer big number.

Description

This function is declared in the `ippcp.h` file. The function evaluates the working buffer assigned to the context `IppsBigNumState` and returns the size of the structure to indicate the maximum length of the integer big number that the structure can store.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Get_BN

Extracts the sign and value of the integer big number from the input structure.

Syntax

```
IppStatus ippsGet_BN(IppsBigNumSGN *sgn, int *length, Ipp32u *data, const
    IppsBigNumState *x);
```

Parameters

sgn	Sign of `IppsBigNumState *x`.
length	Array length of the input data.
data	Data array.
x	Integer big number of the context `IppsBigNumState`.

Description

This function is declared in the `ippcp.h` file. The function extracts the sign and value of the integer big number from the input structure.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

GetOctString_BN

Converts a positive Big Number into octet String.

Syntax

```
IppStatus ippsGetOctString_BN(const Ipp8u* pOctStr, int strLen, const
    IppsBigNumState* pBN);
```

Parameters

pOctStr	Pointer to the input octet string.
strLen	Octet string length in bytes.
pBN	Pointer to the context of the input Big Number.

Description

The function is declared in the `ippcp.h` file. This function converts a positive Big Number into the octet string.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if specified *pOctStr* is insufficient in length.
`ippStsRangeErr`	Indicates an error condition if Big Number is negative.

Example 5-3 Type a Big Number

```
void Type_BN(const char* pMsg, const IppsBigNumState* pBN){
   // size of Big Number
   int size;
   ippsGetSize_BN(pBN, &size);

   // extract Big Number value and convert it to the string presentation
   Ipp8u* bnValue = new Ipp8u [size*4];
   ippsGetOctString_BN(bnValue, size*4, pBN);

   // type header
   if(pMsg)
      cout <<pMsg;

   // type value
   for(int n=0; n<size*4; n++)
      cout<<hex <<(int)bnValue[n];
   cout <<endl;

   delete [] bnValue;
}
```

Cmp_BN

Compares two Big Numbers.

Syntax

```
IppStatus ippsCmp_BN(const IppsBigNumState *pA, const IppsBigNumState
    *pB, Ipp32u *pRsult);
```

Parameters

pA	Pointer to the context of the Big Number A.
pB	Pointer to the context of the Big Number B.
pRsult	Pointer to the result of the comparison.

Description

The function is declared in ippcp.h file. This function compares Big Numbers A and B and sets up the result according to the following conditions:

- if A==B, then **pResult* = IS_ZERO
- if A > B, then **pResult* = GREATER_THAN_ZERO
- if A < B, then **pResult* = LESS_THAN_ZERO

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

CmpZero_BN

Checks the value of the input data field.

Syntax

```
IppStatus ippsCmpZero_BN(const IppsBigNumState *b, Ipp32u *result);
```

Parameters

b	Integer big number of the data type `IppsBigNumState`.
result	Indicates whether the input integer big number is positive, negative, or zero.

Description

This function is declared in the `ippcp.h` file. The function scans the data field of the input `const IppsBigNumState *b` and returns

- `IS_ZERO` if the value held by `IppsBigNumState *b` is zero
- `GREATER_THAN_ZERO` if the input is more than zero
- `LESS_THAN_ZERO` if the input is less than zero.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

Add_BN

Adds two integer big numbers.

Syntax

```
IppStatus ippsAdd_BN(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *r);
```

Parameters

a	First unsigned integer big number of the data type `IppsBigNumState`.
b	Second unsigned integer big number of the data type `IppsBigNumState`.
r	Addition result.

Description

This function is declared in the `ippcp.h` file. The function adds two integer big numbers regardless of their signs and sizes and returns the result of the operation.

The following pseudo code represents this function:

$(*r) \leftarrow (*a) + (*b)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if the size of *r* is smaller than the resulting data length.

NOTE. The function executes only under the condition that size of `IppsBigNumState *r` is not less than either the length of `IppsBigNumState *a` or that of `IppsBigNumState *b`.

Example 5-4 Add Big Numbers

```
void Add_BN_sample(void){
   // define and set up Big Number A
   const Ipp32u bnuA[] = {0x01234567,0x9abcdeff,0x11223344};
   IppsBigNumState* bnA = New_BN(sizeof(bnuA)/sizeof(Ipp32u));

   // define and set up Big Number B
   const Ipp32u bnuB[] = {0x76543210,0xfedcabee,0x44332211};
   IppsBigNumState* bnB = New_BN(sizeof(bnuB)/sizeof(Ipp32u), bnuB);

   // define Big Number R
   int sizeR = max(sizeof(bnuA), sizeof(bnuB));
   IppsBigNumState* bnR = New_BN(1+sizeR/sizeof(Ipp32u));
```

Example 5-4 Add Big Numbers (continued)

```
    // R = A+B
    ippsAdd_BN(bnA, bnB, bnR);

    // type R
    Type_BN("R=A+B:\n", bnR);

    delete [] (Ipp8u*)bnA;
    delete [] (Ipp8u*)bnB;
    delete [] (Ipp8u*)bnR;
}
```

Sub_BN

Subtracts one integer big number from another.

Syntax

```
IppStatus ippsSub_BN(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *r);
```

Parameters

a	First unsigned integer big number of the data type `IppsBigNumState`.
b	Second unsigned integer big number of the data type `IppsBigNumState`.
r	Subtraction result.

Description

This function is declared in the `ippcp.h` file. The function subtracts one integer big number from another regardless of their signs and sizes and returns the result of the operation.

The following pseudo code represents this function:

$(*r) \leftarrow (*a) - (*b)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is smaller than the result data length.

NOTE. The function executes only under the condition that size of `IppsBigNumState *r` is not less than either the length of `IppsBigNumState *a` or that of `IppsBigNumState *b`.

Mul_BN

Multiplies two integer big numbers.

Syntax

```
IppStatus ippsMul_BN(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *r);
```

Parameters

a	Multiplicand of `IppsBigNumState`.
b	Multiplier of `IppsBigNumState`.
r	Multiplication result.

Description

This function is declared in the `ippcp.h` file. The function multiplies an integer big number by another integer big number regardless of their signs and sizes and returns the result of the operation.

The following pseudo code represents this function:

$r \leftarrow a*b$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is smaller than the result data length.

NOTE. The function executes only under the condition that the size `IppsBigNumState *r` is not less than the sum of the lengths of `IppsBigNumState *a` or that of `IppsBigNumState *b` minus one.

MAC_BN_I

Multiplies two integer big numbers and accumulates the result with the third integer big number.

Syntax

```
IppStatus ippsMAC_BN_I(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *r);
```

Parameters

a	Multiplicand of `IppsBigNumState`.
b	Multiplier of `IppsBigNumState`.
r	Multiplication result.

Description

This function is declared in the `ippcp.h` file. The function multiplies one integer big number by another and accumulates the result with the third input integer big number regardless of their signs and sizes. The function subsequently returns the result of the operation.

The following pseudocode represents this function:

$r \leftarrow r + a*b$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is smaller than the result data length.

NOTE. The function executes only under the condition that the size `IppsBigNumState *r` is not less than the sum of the lengths of `IppsBigNumState *a` or that of `IppsBigNumState *b` minus one.

Div_BN

Divides one integer big number by another.

Syntax

```
IppStatus ippsDiv_BN(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *q, IppsBigNumState *r);
```

Parameters

a	Dividend of `IppsBigNumState`.
b	Divisor of `IppsBigNumState`.
q	Quotient of `IppsBigNumState`.
r	Remainder of `IppsBigNumState`.

Description

This function is declared in the `ippcp.h` file. The function divides an integer big number dividend by another integer big number regardless of their signs and sizes and returns the quotient of the division and the respective remainder.

The following pseudo code represents this function:

```
q ← a/b
r ← a – b*q
```

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is smaller than the length of `IppsBigNumState *b` or when the size of `IppsBigNumState *q` is smaller than the quotient result data length.
`ippStsDivByZeroErr`	Indicates an error condition if the zero divisor is attempted.

NOTE. The size of `IppsBigNumState *q` should not be less than `(length of *a) - (length of *b) + 1`, and the size of `IppsBigNumState *r` should be no less than the length of `IppsBigNumState *b`.

Mod_BN

Computes modular reduction for input integer big number with respect to specified modulus.

Syntax

```
IppStatus ippsMod_BN(IppsBigNumState *a, IppsBigNumState *m,
    IppsBigNumState *r);
```

Parameters

a	Integer big number of `IppsBigNumState`.
m	Modulus integer of `IppsBigNumState`.
r	Modular reduction result.

Description

This function is declared in the `ippcp.h` file. The function computes the modular reduction for an input integer big number with respect to the modulus specified by a positive integer big number and returns the modular reduction result in the range of `[0, (m-1)]`.

The following pseudo code represents this function:

$r \leftarrow a \bmod m$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is smaller than the length of `IppsBigNumState *m`.
`ippStsBadModulusErr`	Indicates an error condition if the modulus `IppsBigNumState *m` is not a positive integer.

NOTE. The size of `IppsBigNumState *r` should not be less than the length of `IppsBigNumState *m`.

Gcd_BN

Computes greatest common divisor.

Syntax

```
IppStatus ippsGcd_BN(IppsBigNumState *a, IppsBigNumState *b,
    IppsBigNumState *g);
```

Parameters

a	First integer big number of `IppsBigNumState`.
b	Second integer big number of `IppsBigNumState`.

g Greatest common divisor to *a* and *b*.

Description

This function is declared in the `ippcp.h` file. The function computes the greatest common divisor (GCD) for two positive integer big numbers.

The following pseudo code represents this function:

$g \leftarrow \gcd(a, b)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *g` is smaller than the length of `IppsBigNumState *a` or `IppsBigNumState *b`.

NOTE. The size of `IppsBigNumState *g` should not be less than either the length of `IppsBigNumState *a` and `IppsBigNumState *b`.

ModInv_BN

Computes multiplicative inverse of a positive integer big number with respect to specified modulus.

Syntax

```
IppStatus ippsModInv_BN(IppsBigNumState *e, IppsBigNumState *m,
    IppsBigNumState *d);
```

Parameters

e	Integer big number of `IppsBigNumState`.
m	Modulus integer of `IppsBigNumState`.

d	Multiplicative inverse.

Description

This function is declared in the `ippcp.h` file. The function uses the extended Euclidean algorithm to compute the multiplicative inverse of a given positive integer big number *e* with respect to the modulus specified by another positive integer big number *m*, where `gcd(e, m) = 1`.

The following pseudo code represents this function:

compute *d* such that `d*e = 1 mod m`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsBadArgErr`	Indicates an error condition if *e* is less than or equal to 0.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsBadModulusErr`	Indicates an error condition if the modulus `e` is more than *m*, or `gcd (e,m)` is more than 1, or *m* is less than or equal to 0.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *d` is smaller than the length of `IppsBigNumState *m`.

NOTE

NOTE. The size of `IppsBigNumState *d` should not be less than the length of `IppsBigNumState *m`.

Montgomery Reduction Scheme Functions

This section describes Montgomery reduction scheme functions. The full list of these functions is given in Table 5-2.

Table 5-2 Intel IPP Montgomery Reduction Scheme Functions

Function Base Name	Operation
`MontGetSize`	Gets the size of the `IppsMontState` context.

Table 5-2 Intel IPP Montgomery Reduction Scheme Functions (continued)

Function Base Name	Operation
`MontInit`	Initializes the context and partitions the specified buffer space.
`MontSet`	Sets the input integer big number to a value and computes the Montgomery reduction index.
`MontGet`	Extracts the big number modulus.
`MontForm`	Converts input positive integer big number into Montgomery form.
`MontMul`	Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.
`MontExp`	Computes Montgomery exponentiation.

Montgomery reduction is a technique for efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step.

This section describes functions for Montgomery modular reduction, Montgomery modular multiplication, and Montgomery modular exponentiation.

Let n be a positive integer, and let R and T be integers such that $R > n$, $\gcd(n, R) = 1$, and $0 < T < nR$. The Montgomery reduction of T modulo n with respect to R is defined as $TR - 1 \bmod n$.

For better results, functions included in the cryptography package use $R = b^k$ where $b = 2^{32}$ and k is the Montgomery index integer computed by the ceiling function of the bit length of the integer n over 32.

All functions use employ the context `IppsMontState` to serve as an operational vehicle to carry the Montgomery reduction index k, the integer big number modulus n, the least significant word $n0$ of the multiplicative inverse of the modulus n with respect to the Montgomery reduction factor R, and a sufficient working buffer reserved for various Montgomery modular operations.

Furthermore, two new terms are introduced in this section:

- length of the context `IppsMontState` is defined as the data length of the modulus n carried by the structure
- size of the context `IppsMontState` is therefore defined as the maximum data length of such an integer modulus n that could be carried by this operational vehicle.

The following example can briefly illustrate the procedure of using the primitives described in this section to compute a classical modular exponentiation $T = x^e \bmod n$. Consider computing $T = x^4 \bmod n$, for some integer x with $0 < x < n$.

First get the buffer size required to configure the context `IppsMontState` by calling `MontGetSize` and then allocate the working buffer using OS service function, with allocated buffer to call `MontInit` to initialize the context `IppsMontState`.

Set the modulus n by calling `MontSet` and then convert x into its respective Montgomery form by calling `MontForm`, that is, computing $\underline{x} = xR \bmod n$. Then compute the Montgomery reduction of $\underline{x}\underline{x}$ using the function `MontMul` to generate $T = \underline{x}\underline{x}R^{-1} \bmod n$. The Montgomery reduction of $T*T \bmod n$ with respect to R is $T^2R^{-1} \bmod n = (\underline{x}^2R^{-1})^2R^{-1} \bmod n = x^4R \bmod n$.

Further applying `MontMul` with this value and the value of 1 yields the desired result $T = x^4 \bmod n$.

The classical modular exponentiation should be computed by performing the following sequence of operations:

1. Get the buffer size required to configure the context `IppsMontState` by calling the function `MontGetSize`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer size significantly while using sliding window method enhances the performance.
2. Allocate working buffer through an operating system memory allocation function and configure the structure `IppsMontState` by calling the function `MontInit` with the allocated buffer and the choice made on the modular exponential method at time invoking `MontGetSize`.
3. Call the function `MontSet` to set the integer big number module for `IppsMontState`.
4. Call the function `MontForm` to convert the integer x to be its Montgomery form.
5. Call the function `MontExp` to compute the Montgomery modular exponentiation.
6. Call the function `MontMul` to compute the Montgomery modular multiplication of the above result with the integer 1 as to convert the above result back to the desired classical modular exponential result.
7. Free the memory using an operating system memory free function, if needed.

MontGetSize

Gets the size of the `IppsMontState` *context.*

Syntax

```
IppStatus ippsMontGetSize(IppsExpMethod method, int length, int *size);
```

Parameters

method	Selected exponential method.
length	Data field length for the modulus.
size	Size of the buffer required for initialization.

Description

This function is declared in the `ippcp.h` file. The function specifies the buffer size required to define the structuralized working buffer of the context `IppsMontState` to store the modulus and perform operations using various Montgomery modulus schemes.

The function returns the required buffer size based on the selected exponential method. The binary method helps to significantly reduce the buffer size, while the sliding windows method results in enhanced performance.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.

MontInit

Initializes the context and partitions the specified buffer space.

Syntax

```
IppStatus ippsMontInit(IppsExpMethod method, int length, IppsMontState
    *m);
```

Parameters

method	Selected exponential method.
buffer	Buffer for initializing *m*.
length	Data field length for the modulus.
m	Pointer to the context `IppsMontState`.

Description

This function is declared in the `ippcp.h` file. The function initializes the context using the specified buffer space. The function then partitions the buffer using the selected modular exponential method in such a way as to carry up to *length*sizeof*(Ipp32u)-bit big number modulus and execute various Montgomery modulus operations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.

MontSet

Sets the input integer big number to a value and computes the Montgomery reduction index.

Syntax

```
IppStatus ippsMontSet(const Ipp32u *n, int length, IppsMontState *m);
```

Parameters

n	Input big number modulus.
m	Pointer to the context `IppsMontState` capturing the modulus and the least significant word of the multiplicative inverse *Ni*.

Description

This function is declared in the `ippcp.h` file. The function sets the input positive integer big number *n* to be the modulus for the context `IppsMontState *m`, computes the Montgomery reduction index *k* with respect to the input big number modulus *n* and the least significant 32-bit word of the multiplicative inverse *Ni* with respect to the modulus *R*, that satisfies $R*R^{-1}-n*Ni = 1$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsBadModulusErr`	Indicates an error condition if the modulus is not a positive odd integer.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.
`ippStsOutOfRangeErr`	Indicates an error condition if *length* is larger than `IppsMontState *m`.

MontGet

Extracts the big number modulus.

Syntax

```
IppStatus ippsMontGet(Ipp32u *n, int *length, const IppsMontState *m);
```

Parameters

m	context `IppsMontState`.
n	Modulus data field.
length	Modulus data length.

Description

This function is declared in the `ippcp.h` file. The function extracts the big number modulus from the input `IppsMontState *m`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

MontForm

Converts input positive integer big number into Montgomery form.

Syntax

```
IppStatus ippsMontForm(IppsBigNumState *a, IppsMontState *m,
    IppsBigNumState *r);
```

Parameters

a	Input integer big number within the range `[0, m - 1]`.
m	Input big number modulus of `IppsBigNumState`.

r	Resulting Montgomery form `r = a*R mod m`.

Description

This function is declared in the `ippcp.h` file. The function converts an input positive integer big number into the Montgomery form with respect to the big number modulus and stores the conversion result.

The following pseudo code represents this function:

`r <- a*R mod m.`

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsBadArgErr`	Indicates an error condition if *a* is a negative integer.
`ippStsScaleRangeErr`	Indicates an error condition if *a* is more than *m*.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is larger than `IppsMontState *m`.

NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus *m*.

MontMul

Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.

Syntax

```
IppStatus ippsMontMul(IppsBigNumState *a, IppsBigNumState *b,
    IppsMontState *m, IppsBigNumState *r);
```

Parameters

a	Multiplicand within the range `[0, m - 1]`.
b	Multiplier within the range `[0, m - 1]`.
m	Modulus.
r	Montgomery multiplication result.

Description

This function is declared in the `ippcp.h` file. The function computes the Montgomery modular multiplication for positive integer big numbers of Montgomery form with respect to the modulus `IppsMontState *m`. As a result, `IppsBigNumState *r` holds the product.

The following pseudo code represents this function:

$r \leftarrow a*b*R^{-1} \bmod m.$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsBadArgErr`	Indicates an error condition if *a* or *b* is a negative integer.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsScaleRangeErr`	Indicates an error condition if *a* or *b* is more than *m*.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is larger than `IppsMontState *m`.

NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus *m*.

Example 5-5 Montgomery Multiplication

```
void MontMul_sample(void){
   int size;
   // define and initialize Montgomery Engine over Modulus N
   Ipp32u bnuN = 19;
   ippsMontGetSize(IppsBinaryMethod, 1, &size);
   IppsMontState* pMont = (IppsMontState*)( new Ipp8u [size] );
   ippsMontInit(IppsBinaryMethod, 1, pMont);
   ippsMontSet(&bnuN, 1, pMont);

   // define and init Big Number multiplicant A
   Ipp32u bnuA = 12;
   IppsBigNumState* bnA = New_BN(1, &bnuA);
   // encode A into Montfomery form
   ippsMontForm(bnA, pMont, bnA);

   // define and init Big Number multiplicant A
   Ipp32u bnuB = 15;
   IppsBigNumState* bnB = New_BN(1, &bnuB);

   // compute R = A*B mod N
   IppsBigNumState* bnR = New_BN(1);
   ippsMontMul(bnA, bnB, pMont, bnR);

   Type_BN("R = A*B mod N:\n", bnR);

   delete [] (Ipp8u*)pMont;
   delete [] (Ipp8u*)bnA;
   delete [] (Ipp8u*)bnB;
   delete [] (Ipp8u*)bnR;
}
```

MontExp

Computes Montgomery exponentiation.

Syntax

```
IppStatus ippsMontExp(IppsBigNumState *a, IppsBigNumState *e,
    IppsMontState *m, IppsBigNumState *r);
```

Parameters

a	Big number Montgomery integer within the range of [0, *m* - 1].
e	Big number exponent.
m	Modulus.
r	Montgomery exponentiation result.

Description

This function is declared in the `ippcp.h` file. The function computes Montgomery exponentiation with the exponent specified by the input positive integer big number to the given positive integer big number of the Montgomery form with respect to the modulus *m*.

The following pseudo code represents this function:

$r \leftarrow a^e R^{-(e-1)} \bmod m.$

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsBadArgErr`	Indicates an error condition if *a* or *e* is a negative integer.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsScaleRangeErr`	Indicates an error condition if *a* or *e* is more than *m*.
`ippStsOutOfRangeErr`	Indicates an error condition if `IppsBigNumState *r` is larger than `IppsMontState *m`.

NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus *m*.

Pseudorandom Number Generation Functions

Many cryptographic systems rely on pseudorandom number generation functions in their design that make the unpredictable nature inherited from a pseudorandom number generator the security foundation to ensure safe communication over open channels and protection against potential adversaries.

The full list of Intel IPP Pseudorandom Number Generation Functions is given in Table 5-3.

Table 5-3 Intel IPP Pseudorandom Number Generation Functions

Function Base Name	Operation
Pseudorandom Number Generation Functions	
`PRNGGetSize`	Gets the size of the `IppsPRNGState` context.
`PRNGInit`	Initializes user supplied memory as `IppsPRNGState` context for future use.
`PRNGSetSeed`	Sets the initial state with the given input seed for pseudorandom number generation.
`PRNGSetAugment`	Sets the initial state with the given input entropy for the pseudorandom number generation.
`PRNGSetModulus`	Sets the initial state with the given input modulus for the pseudorandom number generation.
`PRNGSetH0`	Sets the initial state with the given input IV for the SHA-1 algorithm.
`PRNGen`	Generates a pseudorandom unsigned Big Number of the specified bitlength.
`PRNGen_BN`	Generates a pseudorandom positive Big Number of the specified bitlength.

This section describes functions that comprise the pseudorandom bit sequence generator implemented by a US FIPS-approved method and based on a SHA-1 one-way hash function specified by [FIPS PUB 186-2], appendix 3.

The application code for generating a sequence of pseudorandom bits should perform the following sequence of operations:

1. Call the function `PRNGGetSize` to get the size required to configure the `IppsPRNGState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `PRNGInit` function to set up the default value of the parameters for pseudorandom generation process.
3. If the default values of the parameters are not satisfied, call the function `PRNGSetSeed` and/or `PRNGSetArgument` and/or `PRNGSetModulus` and/or `PRNGSetH0` to reset any of the control pseudorandom generator parameters.
4. Keep calling the function `PRNGGen` or `PRNGGen_BN` to generate pseudo random value of the desired format.
5. Free the memory allocated for the `IppsPRNGState` context by calling the operating system memory free service function.

User's Implementation of a Pseudorandom Number Generator

Both functions `ippsPRNGGen` and `ippsPRNGGen_BN` as well as their supplementary functions represent the implementation of the pseudorandom number generator in the IPPCP library. This given implementation is based on recommendations made in [FIPS PUB 186-2]. If you prefer to use the implementation of the pseudorandom number generator which is different from the given, you can still use IPPCP library. To do this, use the following definition of the generator introduced by the IPPCP library:

Syntax

```
typedef IppStatus(_STDCALL *IppBitSupplier)(Ipp32u* pData, int nBits,
    void* pEbsParams);
```

Parameters

pData	Pointer to the output data.
nBits	Number of generated data bits.
pEbsParams	Pointer to the user defined context.

Description

This declaration is included in the ippcp.h file. The function generates any data (probably pseudorandom numbers) of the specified *nBits* length.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsErr	Indicates an error condition.

PRNGGetSize

Gets the size of the IppsPRNGState *context in bytes.*

Syntax

```
IppStatus ippsPRNGGetSize(int *pSize);
```

Parameters

pSize	Pointer to the IppsPRNGState context size in bytes.

Description

This function is declared in the ippcp.h file. The function gets the IppsPRNGState context size in bytes and stores it in **pSize*.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.

PRNGInit

Initializes user supplied memory as `IppsPRNGState` *context for future use.*

Syntax

```
IppStatus ippsPRNGInit(int seedBits, IppsPRNGState* pCtx);
```

Parameters

seedBits	Size in bits for the seed value.
pCtx	Pointer to the `IppsPRNGState` context being intialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsPRNGState` context. In addition, the function sets up the default internal random generator parameters (seed, entropy augment, modulus, and initial hash value H0 of the SHA-1 algorithm). PRNG default parameters are as follows:

- seed =0x0
- entropy augment =0x0
- modulus =0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
- H0 =0xC3D2E1F01032547698BADCFEEFCDAB8967452301

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *seedBits* is less than 1 or greater than 512.

PRNGSetSeed

Sets up the seed value for the pseudorandom number generator.

Syntax

```
IppStatus ippsPRNGSetSeed(const IppsBigNumState* pSeed, IppsPRNGState*
    pCtx);
```

Parameters

pSeed	Pointer to the seed value being set up.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function resets the seed value with the supplied value of *`seedBits`* bit length.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

NOTE:

NOTE. This function restarts the pseudorandom number generation process, which results in losing already generated pseudorandom numbers.

PRNGSetAugment

Sets the initial state with the given input entropy for the pseudorandom number generation.

Syntax

```
IppsStatus ippsPRNGSetAugment(const IppsBigNumState* pAugment,
    IppsPRNGState* pCtx);
```

Parameters

pAugment	Pointer to the entropy augment value being set up.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function resets entropy augment value with the supplied value of the *`seedBits`* bit length.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

PRNGSetModulus

Sets the initial state with the given input modulus for the pseudorandom number generation.

Syntax

```
IppStatus ippsPRNGSetModulus(const IppsBigNumState* pModulus,
    IppsPRNGState* pCtx);
```

Parameters

pModulus	Pointer to the modulus value being set up.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function resets the modulus value with the supplied value up to 160 bit length.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

PRNGSetH0

Sets the initial state with the given input IV for the SHA-1 algorithm.

Syntax

```
IppStatus ippsPRNGSetH0(const IppsBigNumState* pH0, IppsPRNGState* pCtx);
```

Parameters

pH0	Pointer to the initial hash value being set up.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function resets the initial hash value with the supplied value up to 160 bit length.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

PRNGen

Generates a pseudorandom unsigned Big Number of the specified bitlength.

Syntax

```
IppStatus ippsPRNGen(Ipp32u* pRandBNU, int nBits, void* pCtx);
```

Parameters

pRandBNU	Pointer to the output pseudorandom unsigned integer big number.
nBits	Number of the generated pseudorandom bit.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function generates pseudorandom unsigned integer big number of the specified *nBits* length.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsLengthErr`	Indicates an error condition if *nBits* is less than 1.

PRNGen_BN

Generates a pseudorandom positive Big Number of the specified bitlength.

Syntax

```
IppStatus ippsPRNGen_BN(IppsBigNumState* pRandBN, int nBits, void* pCtx);
```

Parameters

pRandBN	Pointer to the output pseudorandom Big Number.
nBits	Number of the generated pseudorandom bit.
pCtx	Pointer to the `IppsPRNGState` context.

Description

This function is declared in the `ippcp.h` file. The function generates pseudorandom positive Big Number of the specified *nBits* length.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.
ippStsLengthErr	Indicates an error condition if *nBits* is less than 1.

Example 5-6 Find Pseudorandom Co-primes

```
void FindCoPrimes(void){
   int size;

   // define Pseudo Random Generator (default settings)
   ippsPRNGGetSize(&size);
   IppsPRNGState* pPrng = (IppsPRNGState*)(new Ipp8u [size] );
   ippsPRNGInit(160, pPrng);

   // define 256-bits Big Numbers X and Y
   const int bnBitSize = 256;
   IppsBigNumState* bnX = New_BN(bnBitSize/32);
   IppsBigNumState* bnY = New_BN(bnBitSize/32);

   // define temporary Big Numbers GCD and 1
   IppsBigNumState* bnGCD = New_BN(bnBitSize/32);
   Ipp32u one = 1;
   IppsBigNumState* bnOne = New_BN(1, &one);
```

Example 5-6 Find Pseudorandom Co-primes (continued)

```
// generate pseudo random X and Y
   // while GCD(X,Y) != 1
   Ipp32u result;
   int counter;
   for(counter=0,result=1; result; counter++) {
      ippsPRNGen_BN(bnX, bnBitSize, pPrng);
      ippsPRNGen_BN(bnY, bnBitSize, pPrng);
      ippsGcd_BN(bnX, bnY, bnGCD);
      ippsCmp_BN(bnGCD, bnOne, &result);
   }
cout <<"Coprimes:" <<endl;
   Type_BN("X: ", bnX); cout <<endl;
   Type_BN("Y: ", bnY); cout <<endl;
   cout <<"were fond on " <<counter <<" attempt" <<endl;

   delete [] (Ipp8u*)pPrng;
   delete [] (Ipp8u*)bnX;
   delete [] (Ipp8u*)bnY;
   delete [] (Ipp8u*)bnGCD;
   delete [] (Ipp8u*)bnOne;
}
```

Prime Number Generation Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions for prime number generation.

The full list of prime number generation functions is given in Table 5-4.

Table 5-4 Intel IPP Prime Number Generation Functions

Function Base Name	Operation
Prime Number Generation Functions	
`PrimeGetSize`	Gets the size of the `IppsPrimeState` context.
`PrimeInit`	Initializes user supplied memory as the `IppsPrimeState` context for future use.
`PrimeGen`	Generates a random probable prime number of the specified bitlength.
`PrimeTest`	Tests the given integer for being a probable prime.
`PrimeSet`	Sets the Big Number for primality testing.
`PrimeSet_BN`	Sets the Big Number for primality testing.
`PrimeGet`	Extracts the probable prime unsigned integer big number.
`PrimeGet_BN`	Extracts the probable prime positive Big Number.

This section describes Intel IPP functions for generating probable prime numbers of variable lengths and validating probable prime numbers through a probabilistic primality test scheme for cryptographic use. A probable prime number is thus defined as an integer that passes the Miller-Rabin probabilistic primality-based test.

The scheme adopted for the probable prime number generation is based on a well-known prime number theorem. Study shows that the number of primitives that are no greater than the given large integer x is closely approximated by the expression. Let denote the number of primes that are not greater than x. In this case the statement is true

$$\lim_{x \to \infty} \frac{\pi(x)}{x/(\ln x)} = 1 .$$

Further study indicates that if X represents the event where the tested k-bit integer n is composite and if Y_t denotes the event where the Miller-Rabin test with the security parameter t declares n to be a prime, the test error probability is upper bounded by $P_{k,t} \le k^{2/3} 2^{t-1/2} t 4^{2-\sqrt{tk}}$ for $t = 2$, $k \ge 88$, or $3 \le t \le k/9$, $k \ge 21$.

Subsequently, a practical strategy for generating a random *k*-bit probable prime is to repeatedly pick *k*-bit random odd integers until finding one integer that can pass a recognized probabilistic primality test scheme as a probable prime. The available set of probable prime number generation functions enables you to specify an appropriate value of the security parameter *t* used in the Miller-Rabin primality test to meet the cryptographic requirements for your application.

All Intel IPP for prime number generation use the context `IppsPrimeState` as an operational vehicle that carries the bitlength of the target probable prime number, the structure capturing the state of the pseudorandom number generation, the structuralized working buffer used for Montgomery modular computation in the Miller-Rabin primality test, and the buffer to store the generated probable prime number.

The following sequence of operations is required to generate a probable prime number of the specified bitlength:

1. Call the function `PrimeGetSize` to get the size required to configure the `IppsPrimeState` context.
2. Allocate memory through the operating system memory allocation function and configure the `IppsPrimeState` context by calling the function `PrimeInit`.
3. Generate probable prime number of the specified bitlength by calling the function `PrimeGen`. If the returned `IppStatus` is `ippStsInsufficientEntropy`, then change the parameters of the pseudorandom generator and call the function `PrimeGen` again.
4. Extract the generated probable prime number by calling the functions `PrimeGet` and `PrimeGet_BN`.
5. Free the memory allocated to the `IppsPrimeState` context by calling the operating system memory-free service function.

PrimeGetSize

Gets the size of the `IppsPrimeState` *context in bytes.*

Syntax

```
IppStatus ippsPrimeGetSize(int nMaxBits, int* pSize);
```

Parameters

nMaxBits	Maximum length of the probable prime number in bits.

pSize	Pointer to the `IppsPrimeState` context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsPrimeState` context size in bytes and stores it in *`pSize`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`nMaxBits`* is less than 1.

PrimeInit

Initializes user supplied memory as `IppsPrimeState` *context for future use.*

Syntax

```
IppStatus ippsPrimeInit(int nMaxBits, IppsPrimeState* pCtx);
```

Parameters

nMaxBits	Maximum length of the probable prime number in bits.
pCtx	Pointer to the `IppsPrimeState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *`pCtx`* as the `IppsPrimeState` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *`nMaxBits`* is less than 1.

PrimeGen

Generates a random probable prime number of the specified bitlength.

Syntax

```
IppStatus ippsPrimeGen(int nBits, int nTrials, IppsPrimeState* pCtx,
    IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

nbits	Target bitlength for the desired probable prime number.
nTrials	Security parameter specified for the Miller-Rabin probable primality.
pCtx	Pointer to the `IppsPrimeState` context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function employs the `rndFuncRandom` Generator specified by the user to generate a random probable prime number of the specified *nBits* length. The generated probable prime number is further validated by the Miller-Rabin primality test scheme with the specified security parameter *nTrials*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *nBits* is less than 1.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if *nTrials* is less than 1.
`ippStsOutOfRangeErr`	Indicates an error condition if *nBits* > *nMaxBits* (see `PrimeGetSize` and `PrimeInit`)

`ippStsInsufficientEntropy`	Indicates a warning condition if prime generation fails due to poor choice of entropy.

PrimeTest

Tests the given integer for being a probable prime.

Syntax

```
IppStatus ippsPrimeTest(int nTrials, Ipp32u *pResult, IppsPrimeState*
    pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

nTrials	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the result of the primality test.
pCtx	Pointer to the `IppsPrimeState` context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function uses the Miller-Rabin probabilistic primality test scheme with the given security parameter to test if the given integer is a probable prime. The pseudorandom number used in the Miller-Rabin test is generated by the specified `rndFunc` Random Generator. The function sets up the **pResult* as IS_PRIME or IS_COMPOSITE to show if the input probable prime passes the Miller-Rabin test.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if *nTrials* is less than 1.

PrimeSet

Sets the Big Number for primality testing.

Syntax

```
IppStatus ippsPrimeSet(const Ipp32u* pBNU, int nBits, IppsPrimeState* pCtx);
```

Parameters

pBNU	Pointer to the unsigned integer big number.
nBits	Unsigned integer big number length in bits.
pCtx	Pointer to the `IppsPrimeState` context.

Description

This function is declared in the `ippcp.h` file. The function sets a probable prime number and its length for the probabilistic primality test.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsLengthErr`	Indicates an error condition if *nBits* is less than 1.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if *nBits* is too large to fit *pCtx*.

PrimeSet_BN

Sets the Big Number for primality testing.

Syntax

```
IppStatus ippsPrimeSet_BN(const IppsBigNumState* pBN, IppsPrimeState* pCtx);
```

Parameters

pBN	Pointer to the Big Number context.
pCtx	Pointer to the `IppsPrimeState` context.

Description

This function is declared in the `ippcp.h` file. The function sets the Big NUmber for probabilistic primality test.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the Big Number is too large to fit *pCtx*.

PrimeGet

Extracts the probable prime unsigned integer big number.

Syntax

```
IppStatus ippsPrimeGet(Ipp32u* pBNU, int *pSize, const IppsPrimeState *pCtx);
```

Parameters

pBNU	Pointer to the unsigned integer big number.
pSize	Pointer to the length of the unsinged integer big number.
p	Pointer to the `IppsPrimeState` context.

Description

This function is declared in the `ippcp.h` file. The function extracts the probable prime number from **pCtx* context and stores it into the specified unsigned integer big number.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

PrimeGet_BN

Extracts the probable prime positive Big Number.

Syntax

```
IppStatus IppsPrimeGet_BN(IppsBigNumState* pBN, const IppsPrimeState *pCtx);
```

Parameters

pBN	Pointer to the Big NUmber context.
pCtx	Pointer to the `IppsPrimeState` context.

Description

This function is declared in the `ippcp.h` file. The function extracts the probable prime positive big number from the **pCtx* context and stores it into the specified Big Number context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the Big Number is too small to store probable prime number.

Example 5-7 Check Primality

```
void CheckPrime(void){
   Ipp32u result;
   int size;

   // define 256-bit Prime Generator
   int maxBitSize = 256;
   ippsPrimeGetSize(maxBitSize, &size);
   IppsPrimeState* pPrimeGen = (IppsPrimeState*)( new Ipp8u [size] );
   ippsPrimeInit(maxBitSize, pPrimeGen);

   // define Pseudo Random Generator (default settings)
   ippsPRNGGetSize(&size);
   IppsPRNGState* pPrng = (IppsPRNGState*)(new Ipp8u [size] );
   ippsPRNGInit(160, pPrng);

   // define known prime value (2^128 -3)/76439
   Ipp32u bnuPrime1[] = {
      0xBEAD208B, 0x5E668076, 0x2ABF62E3, 0xDB7C};
   IppsBigNumState* bnP1 = New_BN(4, bnuPrime1);
   // make sure P1 is really prime
   ippsPrimeSet_BN(bnP1, pPrimeGen);
   ippsPrimeTest(50, &result, pPrimeGen,
                 ippsPRNGen, pPrng);
   IS_PRIME==result?
      cout <<"Primality of P1 is confirmed\n" :
      cout <<"Primality of P1 isn't confirmed\n";
```

Example 5-7 Check Primality (continued)

```
        // define another known prime value 2^128 -2^97 -1
    Ipp32u bnuPrime2[] = {
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFD};
    IppsBigNumState* bnP2 = New_BN(4, bnuPrime2);
    // make sure P2 is really prime
    ippsPrimeSet_BN(bnP2, pPrimeGen);
    ippsPrimeTest(50, &result, pPrimeGen,
                  ippsPRNGen, pPrng);
    IS_PRIME==result?
        cout <<"Primality of P2 is confirmed\n" :
        cout <<"Primality of P2 isn't confirmed\n";

    // define composite Big Number C = P1*P2
    IppsBigNumState* bnC = New_BN(8);
    ippsMul_BN(bnP1, bnP2, bnC);
    // make sure C is really composite
    ippsPrimeSet_BN(bnC, pPrimeGen);
    ippsPrimeTest(50, &result, pPrimeGen,
                  ippsPRNGen, pPrng);
    IS_PRIME==result?
        cout <<"Strange, but C=P1*P2 is prime\n" :
        cout <<"OK, C=P1*P2 is composite\n";

    delete [] (Ipp8u*)pPrimeGen;
    delete [] (Ipp8u*)pPrng;
    delete [] (Ipp8u*)bnP1;
    delete [] (Ipp8u*)bnP2;
    delete [] (Ipp8u*)bnC;
}
```

RSA Algorithm Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions for RSA algorithm. The section describes a set of primitives to perform operations required for RSA cryptographic systems [PKCS 1.2.1]. This set of primitives offers a flexible user interface that enables the RSA crypto key size scalability with the limit of up to 4096 bits.

RSA algorithm functions include

- Functions for Building RSA System, the system being then used by functions listed below.
- RSA Primitives, which perform RSA encryption and decryption.
- RSA-based schemes, which combine RSA cryptographic primitives with other techniques, such as computing hash message digests or applying mask generation functions (MGFs), to achieve a particular security goal. IPPCP contains two groups of functions implementing RSA-based schemes: RSA-OAEP Scheme Functions and RSA-SSA Scheme Functions.

Functions for Building RSA System

The list of functions for building RSA cryptographic system is given in Table 5-5.

Table 5-5 Intel IPP RSA Algorithm Functions

Function Base Name	Operation
`RSAGetSize`	Gets the size of the `IppRSAState` context.
`RSAInit`	Initializes user supplied memory as the `IppRSAState` context for future use.
`RSASetKey`	Sets the tag-designated key component into the established RSA context.
`RSAGetKey`	Extracts the tag-designated key component from the RSA context.
`RSAGenerate`	Generates key components for the desired RSA cryptographic system.
`RSAValidate`	Validates key components of the RSA cryptographic system.

You can use the primitives to build an RSA cryptographic system with the supplied randomized seed and stimulus. The function `RSAGenerate` generates the RSA system probable primes p and q, the system composite integer n, as well as the key pair, the RSA public key e and its respective private key d.

RSA Primitives and RSA-based schemes (RSA-OAEP Scheme Functions and RSA-SSA Scheme Functions) use `IppsRSAState` context, which is initialized using the `RSAInit` function, as an operational vehicle carrying the RSA system composite integer, a pair of RSA probable primes, RSA key pair, and working buffers.

RSAGetSize

Gets the size of the `IppsRSAState` *context.*

Syntax

```
IppStatus ippsRSAGetSize(int nBitsN, int nBitsP, IppsRSAKeyType flag,
    int* pSize);
```

Parameters

nBitsN	Length of the RSA system in bits (that is, the length of the composite RSA modulus n in bits).
nBitsP	Length in bits of the largest of two prime factors of the RSA modulus.
flag	The flag indicating RSA system for encryption or decryption operation.
pSize	Pointer to the IppsRSAState context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRSAState` context size in bytes and stores it in *`*pSize`*. Use the *`flag`* == `IppRSApublic` to establish RSA for encryption or *`flag`* == `IppRSAprivate` to establish RSA for the decryption operation. Refer to RSA-OAEP Scheme Functions and RSA-SSA Scheme Functions on which *`flag`* value to select for the schemes.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsNotSupportedModeErr`	Indicates an error condition if *nBitsN* < 32 or *nBitsN* > 4096.

`ippStsBadArgErr`	Indicates an error condition if (*nBitsP* ≥ *nBitsN*) or (*2*nBitsP* < *nBitsN*) or *flag* value is illegal.

RSAInit

Initializes user supplied memory as the `IppsRSAState` *context for future use.*

Syntax

```
IppStatus ippsRSAInit(int nBitsN, int nBitsP, IppsRSAKeyType flag,
    IppsRSAState* pCtx);
```

Parameters

nBitsN	Length of the RSA system in bits (that is, the length of the composite RSA modulus n in bits)
nBitsP	Length in bits of the largest of two prime factors of the RSA modulus.
flag	The flag indicating RSA system for encryption or decryption operation.
pCtx	Pointer to the `IppsRSAState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsRSAState` context. Use the *flag* == `IppRSApublic` to initialize RSA context for encryption or *flag* == `IppRSAprivate` to initialize RSA context for the decryption operation. Refer to RSA-OAEP Scheme Functions and RSA-SSA Scheme Functions on which *flag* value to select for the schemes.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsNotSupportedModeErr`	Indicates an error condition if *nBitsN* < 32 or *nBitsN* > 4096.

`ippStsBadArgErr`	Indicates an error condition if *`nBitsN`* > *`nBitsP`* or illegal *`flag`* value.

RSASetKey

Sets the tag-designated key component into the established RSA context.

Syntax

```
IppStatus ippsRSASetKey(const IppsBigNumState* pBN, ippsRSAKeyTag tag,
    IppsRSAState* pCtx);
```

Parameters

`pBN`	Pointer to the Big Number context presented by key component.
`tag`	The tag of the key component being set up.
`pCtx`	Pointer to the `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function sets the key specified by *`pBN`* to the tag-designated component of the `IppRSAState` context:

- *`tag`* == `IppRSAkeyN`, the function sets up the RSA system composite integer *n*
- *`tag`* == `IppRSAkeyP`, the function sets up the RSA system prime factor *p*
- *`tag`* == `IppRSAkeyQ`, the function sets up the RSA system prime factor *q*
- *`tag`* == `IppRSAkeyE`, the function sets up the RSA system public exponent *e*
- *`tag`* == `IppRSAkeyD`, the function sets up the RSA system public exponent *d*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsLengthErr`	Indicates an error condition if the length of the Big Number specified by *pBN* is less than 1.
`ippStsBadArgErr`	Indicates an error condition if some of the arguments are invalid: Big Number specified by *pBN* is negative, key component specified by *tag* does not match the RSA context.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the length of the Big Number specified by *pBN* is too big to be stored in the `IppsRSAState` context.

RSAGetKey

Extracts the tag-designated key component from the RSA context.

Syntax

```
IppStatus ippsRSAGetKey(IppsBigNumState* pBN, IppsRSAKeyTag tag,
    IppsRSAState* pCtx);
```

Parameters

pBN	Pointer to the output Big Number context.
tag	The tag of the key component being extracted from.
pCtx	Pointer to the `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function extracts the tag-designated key component from the **pCtx* RSA context and stores it in the specified **pBN* Big Number context:

- *tag* == `IppRSAkeyN`, the function extracts the RSA system composite integer *n*
- *tag* == `IppRSAkeyP`, the function extracts the RSA system prime factor *p*
- *tag* == `IppRSAkeyQ`, the function extracts the RSA system prime factor *q*

- *tag* == `IppRSAkeyE`, the function extracts the RSA system public exponent *e*
- *tag* == `IppRSAkeyD`, the function extracts the RSA system public exponent *d*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsBadArgErr`	Indicates an error condition if the key component speicified by tag does not match the RSA context.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the length of the Big Number specified by *`pBN`* is too small to be stored as a key component.

RSAGenerate

Generates key components for the desired RSA cryptographic system.

Syntax

```
IppStatus ippsRSAGenerate(IppsBigNumState* pE, int nBitsN, int nBitsP, int
    nTrials, IppsRSAState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

`pE`	Pointer to the `IppsBigNumState` context of the newly generated RSA public exponent key.
`nBitsN`	Length of the RSA system in bits (that is, the length of the composite RSA modulus *n* in bits)
`nBitsP`	Length in bits of one of the two prime factors of the RSA modulus.
`nTrials`	Security parameter specified for the Miller-Rabin probable primality
`pCtx`	Pointer to the `IppsRSAState` context.

rndFunc	Specified Random Generator
pRndParam	Pointer to the random Generator context.

Description

The function is declared in the `ippcp.h` file. This function generates the desired RSA cryptographic system based on:

- the input **pE* specifying the initial value for searching the RSA public exponent
- input parameters *nBitsN* and *nBitsP* specifying the bit lengths of the composite RSA modulus n and the largest RSA prime factor p respectively.

This function employs the specified *rndFunc* Random Generator to generate a random probable prime numbers p and q.

The function then computes the RSA composite modulus $n = (p*q)$, and the RSA private exponent d, and it further computes all other CRT-related RSA components.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsNotSupportedModeErr`	Indicates an error condition if *nBitsN* < 32 or *nBitsN* > 4096.
`ippStsBadArgErr`	Indicates an error condition if *nBitsN* > *nBitsP* or *nTrials* < 1.
`ippStsInsufficientEntropy`	Indicates a warning condition if prime generation fails due to poor choice of entropy.

RSAValidate

Validates key components of the RSA cryptographic system.

Syntax

```
IppStatus ippsRSAValidate(const IppsBigNumState* pE, int nTrials, Ipp32u*
    pResult, IppsRSAState* pCtx, IppBitSuppler rndFunc, void* pRndParam);
```

Parameters

pE	Pointer to the `IppsBigNumState` context of the RSA public exponent.
nTrails	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the result of validation.
pCtx	Pointer to the `IppsRSAState` context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function validates key components of the RSA cryptographic system and stores the result (`IS_VALID_KEY` or `IS_INVALID_KEY`) of the validation procedure in **pResult*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if *nBitsN* > *nBitsP* or *nTrials* < 1.
`ippStsLengthErr`	Indicates an error condition if the length of the Big Number specified by *pE* is less than 1.
`ippStsOutOfRangeErr`	Indicates an error condition if the length of the Big Number specified by *pE* is too big to be stored in the `IppsRSAState` context.

RSA Primitives

The list of RSA cryptographic primitives is given in Table 5-6.

Table 5-6 Intel IPP RSA Primitives

Function Base Name	Operation
`RSAEncrypt`	Performs the RSA encryption operation.
`RSADecrypt`	Performs the RSA decryption operation.

The application code for conducting a typical RSA encryption must perform the following sequence of operations, starting with building of a crypto system:

1. Call the function `RSAGetSize` to get the size required to configure `IppsRSAState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `RSAInit` function to initialize the context for the RSA encryption.
3. Keep calling the `RSASetKey` to set up RSA public key (n, e).
4. Invoke the `RSAEncrypt` function with the established RSA public key to encode the plaintext into the respective ciphertext.
5. Free the memory allocated for the `IppsRSAState` context by calling the operating system memory free service function.

The typical application code for the RSA decryption must perform the following sequence of operations:

1. Call the function `RSAGetSize` to get the size required to configure `IppsRSAState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `RSAInit` function to initialize the context for the RSA decryption.
3. Establish the RSA private key by means of either `RSAGenerate` function or by key setup function `RSASetKey`. The `RSAGenerate` function computes the private key d with respect to the generated public key e, as well as several other components for applying the CRT. When using `RSASetKey`, you have an option of presenting the private key either as a pair (n,d) or quantiple (p, q, dP, dQ, $qInv$).
4. Invoke the `RSADecrypt` function with the established RSA public key to decode the ciphertext into the respective plaintext.

5. Free the memory allocated for the `IppsRSAState` context by calling the operating system memory free service function.

RSAEncrypt

Performs the RSA encryption operation.

Syntax

```
IppStatus ippsRSAEncrypt(const IppsBigNumState* pX, IppsBigNumState* pY,
    IppsRSAState* pCtx);
```

Parameters

pX	Ppointer to the `IppsBigNumState` context of the plaintext.
pY	Pointer to the `IppsBigNumState` context of the ciphertext.
pCtx	Pointer to the `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function performs the RSA encryption operation.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if: length of the Big Number specified by *pX* is too big length of the Big Number specified by *pY* is too small.

RSADecrypt

Performs the RSA decryption operation.

Syntax

```
IppStatus ippsRSADecrypt(IppsBigNumState* pX, IppsBigNumState* pY,
    IppsRSAState* pCtx);
```

Parameters

pX	Ppointer to the `IppsBigNumState` context of the ciphertext.
pY	Pointer to the `IppsBigNumState` context of the plaintext.
pCtx	Pointer to the `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function performs the RSA encryption operation.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if: length of the Big Number specified by *pX* is too big or length of the Big Number specified by *pY* is too small.

The example below illustrates the use of RSA primitives. The example uses the `BigNumber` class and functions creating some cryptographic contexts, whose source code can be found in Appendix B.

Example 5-8 The Use of RSA Primitives

```
static Ipp32u dataN[] = {
   0x091DBDCB,0x46F8E5FD,
   0xCA2A8F59,0xE2537298,0xF6C1687F,0x527A9A41,0x7B61A51F,0xE0AAB12D,
   0x4598394E,0x8834B245,0x06095374,0xEE6A649D,0xD93A2584,0x3EE6B4B7,
   0xDFC73772,0xAFB8E0A3,0x5B8B807F,0x19719D8A,0x60E1EC46,0x76ED520D,
   0xEB6FCD48,0x61EA48CE,0x035C02AB,0xB8DFBAAF,0x7454F51F,0x40D6B6F0,
   0xD41043A4,0x368D07EE,0x9DA871F7,0x2338AC2B,0x0682CE9C,0xBBF82F09
};
static Ipp32u dataP[] = {
   0x58FB6599,0x7541BA2A,0x459D1F39,0x5B252176,
   0xAA040A2D,0x7E28FAE7,0x6E5D1E3B,0x124EF023,0x3D84F632,0x93B81A9E,
   0xAEF4FDA4,0x99EB9F44,0xA1B56001,0x08810B10,0xB1B9B3C9,0xEECFAE81
};
static Ipp32u dataQ[] = {
   0xAF461503,0xA441E700,0x4D0416A5,0xCE335252,
   0x3204B5CF,0xEA0DA3B4,0x66B42E92,0x9840B416,0x028B9D86,0x5A0F2035,
   0x8866B1D0,0x3F6C42D0,0xAAD1D935,0x341233EA,0x27F453F6,0xC97FB1F0
};
static Ipp32u dataE[] = {0x11};

int RSA_sample(void)
{
   BigNumber P(dataP, sizeof(dataP)/sizeof(dataP[0]));
   BigNumber Q(dataQ, sizeof(dataQ)/sizeof(dataQ[0]));
   BigNumber N = P*Q;
   BigNumber E(dataE, sizeof(dataE)/sizeof(dataE[0]));
```

Example 5-8 The Use of RSA Primitives (continued)

```
IppsRSAState* pRSApub  = newRSA(N.BitSize(), P.BitSize(), IppRSApublic);
IppsRSAState* pRSAprv1 = newRSA(N.BitSize(), P.BitSize(), IppRSAprivate);
IppsRSAState* pRSAprv2 = newRSA(N.BitSize(), P.BitSize(), IppRSAprivate);

// compute private key
BigNumber phi = (P-BigNumber(1))*(Q-BigNumber(1));
BigNumber D = phi.InverseMul(E);

// set up public RSA (N,E)
ippsRSASetKey(N, IppRSAkeyN, pRSApub);
ippsRSASetKey(E, IppRSAkeyE, pRSApub);

// set up private (no CRT) RSA (N, D)
ippsRSASetKey(N, IppRSAkeyN, pRSAprv1);
ippsRSASetKey(D, IppRSAkeyD, pRSAprv1);

// set up private (CRT) RSA (P,Q,D)
ippsRSASetKey(P, IppRSAkeyP, pRSAprv2);
ippsRSASetKey(Q, IppRSAkeyQ, pRSAprv2);
ippsRSASetKey(D, IppRSAkeyD, pRSAprv2);

// validate RSA
IppsPRNGState* pRand = newPRNG();
Ipp32u result;
ippsRSAValidate(E, 50, &result, pRSAprv2, ippsPRNGen, pRand);

if(IS_VALID_KEY!=result) {
   cout <<"validation fail" <<endl;
   return 0;
}
```

Example 5-8 The Use of RSA Primitives (continued)

```
      // validation pass

      // planetext
      Ipp32u dataM[] = {
         0x4D353E2D,0xD2F1B76D,
         0x5281CE32,0x7BC27519,0x2F3AC14F,0x0448DB97,0xD095AEB4,0x82FB3E87,
         0x1BE392F9,0x43581159,0xD5024121,0xB48D2869,0x2BAAD29A,0xA1B7C136,
         0xF47728B4,0x4CDCFE4F,0x839A2DDB,0xFF8AE10E,0x25C9C2B3,0xF93EDCFB,
         0x4626F5AF,0xD7E0B2C0,0xB4251F84,0xC31B2E8B,0xA8F55267,0x5C68F1EE,
         0x26DCD87D,0xCA82310B,0x504B45E2,0x6350E329,0xACE9E300,0x00EB7A19
      };
      BigNumber M(dataM,sizeof(dataM)/sizeof(dataM[0]));

      // encrypt planetext
      BigNumber C(0,N.DwordSize());
      ippsRSAEncrypt(M, C, pRSApub);

      // decrypt ciphertext using pRSAprv1
      BigNumber Z1(0,N.DwordSize());
      ippsRSADecrypt(C, Z1, pRSAprv1);

      // decrypt ciphertext using pRSAprv2
      BigNumber Z2(0,N.DwordSize());
      ippsRSADecrypt(C, Z2, pRSAprv2);

      deleteRSA(pRSApub);
      deleteRSA(pRSAprv1);
      deleteRSA(pRSAprv2);

      return (M==Z1) && (M==Z2);
   }
```

RSA-OAEP Scheme Functions

This subsection describes functions implementing RSA-OAEP encryption scheme, featured in [PKCS 1.2.1].

The full list of these functions is given in Table 5-7.

Table 5-7 Intel IPP RSA-based Encryption Scheme Functions

Function Base Name	Operation
RSAOAEPEncrypt	Carries out the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_MD5	MD5-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA1	SHA-1-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA224	SHA-224-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA256	SHA-256-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA384	SHA-384-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA512	SHA-512-based helper of the RSA-OAEP encryption scheme.
RSAOAEPDecrypt	Carries out the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_MD5	MD5-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA1	SHA-1-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA224	SHA-224-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA256	SHA-256-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA384	SHA-384-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA512	SHA-512-based helper of the RSA-OAEP decryption scheme.

To invoke a function that carries out RSA-OAEP scheme, the `IppsRSAState` context must be initialized (by the `RSAInit` function) with a proper value of the *`flag`* parameter:

- For the RSA-OAEP encryption scheme, *`flag`* == `IppRSApublic`
- For the RSA-OAEP decryption scheme, *`flag`* == `IppRSAprivate`.

RSAOAEPEncrypt

Carries out the RSA-OAEP encryption scheme.

Syntax

```
IppStatus ippsRSAOAEPEncrypt(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState*
    pCtx, IppHash hushFunc, int hashLen, IppMGF mgfFunc);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.
hashFunc	Hash function, which meets the General Definition of a Hash Function given in chapter 3.
hashLen	Length of the hash function output, in octets.
mgfFunc	Mask generation function (MGF), which meets the definition given in section User's Implementation of a Mask Generation Function in chapter 3.

Description

This function is declared in the `ippcp.h` file. The function carries out the RSA-OAEP encryption scheme, defined in [PKCS 1.2.1].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.
`ippStsLengthErr`	Indicates an error condition if the input length parameters do not meet any of the following conditions: $srcLen > 0$, $srcLen > N - 2 \cdot hashLen - 2$, where N is the length of the RSA modulus in octets, $labLen > 0$, $hashLen > 0$.

RSAOAEPEncrypt_MD5

Carries out the RSA-OAEP encryption scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_MD5(const Ipp8u* pSrc, int srcLen, const
    Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst,
    IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPEncrypt` function returns.

RSAOAEPEncrypt_SHA1

Carries out the RSA-OAEP encryption scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_SHA1(const Ipp8u* pSrc, int srcLen, const
    Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst,
    IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general RSAOAEPEncrypt function returns.

RSAOAEPEncrypt_SHA224

Carries out the RSA-OAEP encryption scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_SHA224(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general RSAOAEPEncrypt function returns.

RSAOAEPEncrypt_SHA256

Carries out the RSA-OAEP encryption scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_SHA256(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPEncrypt` function returns.

RSAOAEPEncrypt_SHA384

Carries out the RSA-OAEP encryption scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_SHA384(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPEncrypt` function returns.

RSAOAEPEncrypt_SHA512

Carries out the RSA-OAEP encryption scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPEncrypt_SHA512(const Ipp8u* pSrc, int srcLen, const Ipp8u*
    pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet message to be encrypted.
srcLen	Length of the message to be encrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pSeed	Pointer to the random octet string of length *hashLen*, where *hashLen* is the length (in octets) of the hash message digest.
pDst	Pointer to the output octet ciphertext string.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general `RSAOAEPEncrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPEncrypt` function returns.

RSAOAEPDecrypt

Carries out the RSA-OAEP decryption scheme.

Syntax

```
IppStatus ippsRSAOAEPDecrypt(const Ipp8u* pSrc, const Ipp8u* pLabel, int
    labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx, IppHash
    hushFunc, int hashLen, IppMGF mgfFunc);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.
hashFunc	Hash function, which meets the General Definition of a Hash Function given in chapter 3.
hashLen	Length of the hash function output, in octets.
mgfFunc	Mask generation function (MGF), which meets the definition given in section User's Implementation of a Mask Generation Function in chapter 3.

Description

This function is declared in the `ippcp.h file`. The function carries out the RSA-OAEP decryption scheme defined in [PKCS 1.2.1].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.
`ippStsLengthErr`	Indicates an error condition if the input length parameters do not meet any of the following conditions: $labLen > 0$, $hashLen > 0$, $2 \cdot hashLen + 2 \geq N$, where N is the length of the RSA modulus in octets.L

RSAOAEPDecrypt_MD5

Carries out the RSA-OAEP decryption scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_MD5(const Ipp8u* pSrc, const Ipp8u* pLabel,
    int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA1

Carries out the RSA-OAEP decryption scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_SHA1(const Ipp8u* pSrc, const Ipp8u*
    pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1sbased MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA224

Carries out the RSA-OAEP decryption scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_SHA224(const Ipp8u* pSrc, const Ipp8u*
    pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA256

Carries out the RSA-OAEP decryption scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_SHA256(const Ipp8u* pSrc, const Ipp8u*
    pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA384

Carries out the RSA-OAEP decryption scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_SHA384(const Ipp8u* pSrc, const Ipp8u*
    pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA512

Carries out the RSA-OAEP decryption scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEPDecrypt_SHA512(const Ipp8u* pSrc, const Ipp8u*
    pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general `RSAOAEPDecrypt` function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSA-SSA Scheme Functions

This subsection describes functions implementing RSA-SSA signature scheme with appendix, featured in [PKCS 1.2.1].

The full list of these functions is given in Table 5-8.

Table 5-8 Intel IPP RSA-based Signature Scheme Functions

Function Base Name	Operation
RSASSASign	Carries out the RSA-SSA signature generation scheme.
RSASSASign_MD5	MD5-based helper of the RSA-SSA signature generation scheme.
RSASSASign_SHA1	SHA-1-based helper of the RSA-SSA signature generation scheme.
RSASSASign_SHA224	SHA-224-based helper of the RSA-SSA signature generation scheme.
RSASSASign_SHA256	SHA-256-based helper of the RSA-SSA signature generation scheme.
RSASSASign_SHA384	SHA-384-based helper of the RSA-SSA signature generation scheme.
RSASSASign_SHA512	SHA-512-based helper of the RSA-SSA signature generation scheme.
RSASSAVerify	Carries out the RSA-SSA signature verification scheme.
RSASSAVerify_MD5	MD5 based helper of the RSA-SSA signature verification scheme.
RSASSAVerify_SHA1	SHA-1-based helper of the RSA-SSA signature verification scheme.
RSASSAVerify_SHA224	SHA-224-based helper of the RSA-SSA signature verification scheme.
RSASSAVerify_SHA256	SHA-256-based helper of the RSA-SSA signature verification scheme.
RSASSAVerify_SHA384	SHA-384-based helper of the RSA-SSA signature verification scheme.
RSASSAVerify_SHA512	SHA-512-based helper of the RSA-SSA signature verification scheme.

To invoke a function that carries out RSA-SSA scheme, the `IppsRSAState` context must be initialized (by the `RSAInit` function) with a proper value of the *`flag`* parameter:

- For the RSA-SSA signing scheme, *`flag`* == `IppRSAprivate`
- For the RSA-SSA verification scheme, *`flag`* == `IppRSApublic`.

RSASSASign

Carries out the RSA-SSA signature generation scheme.

Syntax

```
IppStatus ippsRSASSASign(const Ipp8u* pHMsg, int hashLen, const Ipp8u*
    pSalt, int saltLen, Ipp8u* pSign, IppsRSAState* pCtx, IppHash
    hushFunc, IppMGF mgfFunc);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
hashLen	Length of the message hash **pHMsg* in octets.
pSalt	Pointer to the random octet salt string
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.
hashFunc	Hash function, which meets the General Definition of a Hash Function given in chapter 3.
mgfFunc	MGF, which meets the definition given in section User's Implementation of a Mask Generation Function in chapter 3.

Description

This function is declared in the `ippcp.h` file. The function generates a message signature according to the RSASSA-PSS scheme defined in [PKCS 1.2.1]. Intel IPP implementation of the scheme assumes that its first step, i.e. computing the hash digest of the original message, is executed prior to the function call and the resulting message hash **pHMsg* is passed to the function. The use of a message hash instead of the original message reduces the length of the function input message, limited by the upper bound of the integer data type ($(2^{32}-1)\cdot 8$ bit), and thus allows applying the entire RSA-SSA scheme to input messages of greater lengths. To compute the original message hash to be passed to the function, you should use the same hash function as the one specified by the *hashFunc* parameter and applied at the subsequent steps of the scheme.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.

ippStsLengthErr	Indicates an error condition if the input length parameters do not meet any of the following conditions: $hashLen > 0$, $saltLen \geq 0$, $N > hashLen + saltLen + 2$, where N is the length of the RSA modulus in octets.

RSASSASign_MD5

Carries out the RSA-SSA signature generation scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_MD5(const Ipp8u* pHMsg, const Ipp8u* pSalt, int
    saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSASign` function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the `MD5MessageDigest` function to generate the hash message to be signed.
2. Call `RSASSASign_MD5` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general RSASSASign function returns.

RSASSASign_SHA1

Carries out the RSA-SSA signature generation scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA1(const Ipp8u* pHMsg, const Ipp8u* pSalt, int
    saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general RSASSASign function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the SHA1MessageDigest function to generate the hash message to be signed.
2. Call `RSASSASign_SHA1` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general RSASSASign function returns.

RSASSASign_SHA224

Carries out the RSA-SSA signature generation scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA224(const Ipp8u* pHMsg, const Ipp8u* pSalt,
    int saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSASign` function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the `SHA224MessageDigest` function to generate the hash message to be signed.
2. Call `RSASSASign_SHA224` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general `RSASSASign` function returns.

RSASSASign_SHA256

Carries out the RSA-SSA signature generation scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA256(const Ipp8u* pHMsg, const Ipp8u* pSalt,
    int saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSASign` function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the `SHA256MessageDigest` function to generate the hash message to be signed.
2. Call `RSASSASign_SHA256` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general `RSASSASign` function returns.

RSASSASign_SHA384

Carries out the RSA-SSA signature generation scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA384(const Ipp8u* pHMsg, const Ipp8u* pSalt,
    int saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSASign` function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the `SHA384MessageDigest` function to generate the hash message to be signed.
2. Call `RSASSASign_SHA384` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general `RSASSASign` function returns.

RSASSASign_SHA512

Carries out the RSA-SSA signature generation scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA512(const Ipp8u* pHMsg, const Ipp8u* pSalt,
    int saltLen, Ipp8u* pSign, IppsRSAState* pCtx);
```

Parameters

pHMsg	Pointer to the octet message hash to be signed.
pSalt	Pointer to the random octet salt string.
saltLen	Length of the salt string in octets.
pSign	Pointer to the output octet signature.
pCtx	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSASign` function. Provided the `IppsRSAState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the `SHA512MessageDigest` function to generate the hash message to be signed.
2. Call `RSASSASign_SHA512` with *pHMsg* pointing to the resulting message hash.

Return Values

The function may return any of the values that the general `RSASSASign` function returns.

RSASSAVerify

Carries out the RSA-SSA signature verification scheme.

Syntax

```
IppStatus ippsRSASSAVerify(const Ipp8u* pHMsg, int hashLen, const Ipp8u* pSign,
    IppBool* pIsValid, IppsRSAState* pCtx, IppHash hushFunc, IppMGF mgfFunc);
```

Parameters

pHMsg	Pointer to the octet message hash that has been signed.
hashLen	Length in octets of the message hash **pHMsg*.
pSign	Pointer to the octet signature string to be verified.
pIsValid	Pointer to the verification result.
pCtx	Pointer to the properly initialized `IppsRSAState` context.
hashFunc	Hash function, which meets the General Definition of a Hash Function given in chapter 3.
mgfFunc	MGF, which meets the definition given in section User's Implementation of a Mask Generation Function in chapter 3.

Description

This function is declared in the `ippcp.h` file. The function carries out the RSASSA-PSS signature verification scheme defined in [PKCS 1.2.1]].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.

`ippStsLengthErr`	Indicates an error condition if the input length parameter does not meet any of the following conditions: *hashLen* > 0, *hashLen* + 2 > *N*, where *N* is the length of the RSA modulus in octets.

RSASSAVerify_MD5

Carries out the RSA-SSA signature verification scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_MD5(const Ipp8u* pHMsg, const Ipp8u* pSign,
    IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

RSASSAVerify_SHA1

Carries out the RSA-SSA signature verification scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA1(const Ipp8u* pHMsg, const Ipp8u* pSign,
    IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

RSASSAVerify_SHA224

Carries out the RSA-SSA signature verification scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA224(const Ipp8u* pHMsg, const Ipp8u*
    pSign, IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

RSASSAVerify_SHA256

Carries out the RSA-SSA signature verification scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA256(const Ipp8u* pHMsg, const Ipp8u*
    pSign, IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

RSASSAVerify_SHA384

Carries out the RSA-SSA signature verification scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA384(const Ipp8u* pHMsg, const Ipp8u*
    pSign, IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

RSASSAVerify_SHA512

Carries out the RSA-SSA signature verification scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA512(const Ipp8u* pHMsg, const Ipp8u*
    pSign, IppBool* pIsValid, IppsRSAState* pCtx);
```

Parameters

`pHMsg`	Pointer to the octet message hash that has been signed.
`pSign`	Pointer to the octet signature string to be verified.
`pIsValid`	Pointer to the verification result.
`pCtx`	Pointer to the properly initialized `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to verify a signature according to the RSASSA-PSS scheme and thus is a specific form of the general `RSASSAVerify` function.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns.

Discrete-Logarithm-Based Cryptography Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions allowing for different operations with Discrete Logarithm (DL) based cryptosystem over a prime finite field GF(p). The functions are mainly based on the [IEEE P1363A] standard. Implementation of the Digital Signature operations is based on [FIPS PUB 186-2]. The Diffie-Hellman (DH) Agreement scheme is based on [X9.42].

The full list of Intel IPP DL-based cryptography functions is given in Table 5-9.

Table 5-9 Intel IPP Discrete-Logarithm-Based Cryptography Functions

Function Base Name	Operation
DLPGetSize	Gets the size of the IppsDLPState context.
DLPInit	Initializes user supplied memory as the IppsDLPState context for future use.
DLPSet	Sets up domain parameters of the DL-based cryptosystem over GF(*p*).
DLPGet	Retrieves domain parameters of the DL-based cryptosystem over GF(*p*).
DLPSetDP	Sets up a particular domain parameter of the DL-based cryptosystem over GF(*p*).
DLPGetDP	Retrieves a particular domain parameter of the DL-based cryptosystem over GF(*p*).
DLPGenKeyPair	Generates a private key and computes public keys of the DL-based cryptosystem over GF(*p*).
DLPPublicKey	Computes a public key from the given private key of the DL-based cryptosystem over GF(*p*).
DLPValidateKeyPair	Validates private and public keys of the DL-based cryptosystem over GF(*p*).
DLPSetKeyPair	Sets private and/or public keys of the DL-based cryptosystem over GF(*p*).
DLPGenerateDSA	Generates domain parameters of the DL-based cryptosystem over GF(*p*) to use DSA.
DLPValidateDSA	Validates domain parameters of the DL-based cryptosystem over GF(*p*) to use DSA.
DLPSignDSA	Performs the DSA digital signature signing operation.
DLPVerifyDSA	Verifies the input DSA digital signature.
DLPGenerateDH	Generates domain parameters of the DL-based cryptosystem over GF(*p*) to use the DH Agreement scheme.
DLPValidateDH	Validates domain parameters of the DL-based cryptosystem over GF(*p*) to use the DH Agreement scheme.
DLPSharedSecretDH	Computes a shares secret field element by using the Diffie-Hellman scheme.

All functions described in this section employ the IppsDLPState context as operational vehicle that carries domain parameters of the DL cryptosystem, a pair of keys, and working buffers.

The application code intended for executing typical operations should perform the following sequence of operations:

1. Call the function `DLPGetSize` to get the size required to configure the `IppsDLPState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `DLPInit` function to initialize the context of the DL-based cryptosystem.
3. Set domain parameters of the DL-based cryptosystem by calling the `DLPSet` function, or generate domain parameters by calling the `DLPGenerateDSA` or `DLPGenerateDH`.
4. Call one of the functions `DLPSignDSA`, `DLPVerifyDSA`, and `DLPSharedSecretDH` to compute digital signature, to verify authenticity of the digital signature, and to compute the shared element accordingly.
5. Free the memory allocated for the `IppsDLPState` context by calling the operating system memory free service function unless the context is no longer needed.

Free the memory allocated for the `IppsPRNGState` context by calling the operating system memory free service function.

DLPGetSize

Gets the size of the `IppsDLPState` *context.*

Syntax

```
IppStatus ippsDLPGetSize(int peBits, int reBits, int *pSize);
```

Parameters

peBits	Bitsize of the GF(*p*) element (that is, the length of the DL-based cryptosystem in bits)
reBits	Bitsize of the multiplicative subgroup GF(*r*).
pSize	Pointer to the `IppsDLPState` context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsDLPState` context size in bytes and stores in `*pSize`. DL-based cryptosystem over GF(*p*) assumes that *r/p*-1 where both *p* and *r* are primes.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if *peBits* ≤ *reBits*.

DLPInit

Initializes user supplied memory as the `IppsDLPState` *context for future use.*

Syntax

```
IppsStatus IppsDLPInit(int peBits, int reBits, IppsDLPState* pCtx);
```

Parameters

peBits	Bitsize of the GF(p) element (that is, the length of the DL-based cryptosystem in bits)
reBits	Bitsize of the multiplicative subgroup GF(r).
pCtx	Pointer to the `IppsDLPState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDLPState` context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if *peBits* ≤ *reBits*.

DLPSet

Sets up domain parameters of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPSet(const IppsBigNumState* pP, const IppBigNumState*
    pQ, const IppsBigNumState* pG, IppsDLPState* pCtx);
```

Parameters

pP	Pointer to the characteristic p of the prime finite field GF(p).
pQ	Pointer to the characteristic q of the multiplicative subgroup GF(q).
pG	Pointer to the generator G of the multiplicative subgroup GF(r).
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function sets up DL-based cryptosystem domain parameters into the cryptosystem context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsRangeErr`	Indicates an error condition if any of the Big Numbers specified by *pP*, *pR*, and *pG* is too big to be stored in the `IppsDLPState` context.

DLPGet

Retrieves domain parameters of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPGet(IppsBigNumState* pP, IppsBigNumState* pQ,
    IppsBigNumState* pG, ippsDLPState* pCtx);
```

Parameters

pP	Pointer to the characteristic p of the prime finite field GF(p).
pQ	Pointer to the characteristic q of the multiplicative subgroup GF(q).
pG	Pointer to the generator G of the multiplicative subgroup GF(r).
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function retrieves DL-based cryptosystem domain parameters into the cryptosystem context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsRangeErr`	Indicates an error condition if any of the Big Numbers specified by *pP*, *pR*, and *pG* is too small for the DL parameter.

DLPSetDP

Sets up a particular domain parameter of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPSetDP(const IppsBigNumState* pDP, IppsDLPKeyTag tag,
    IppsDLPState* pCtx);
```

Parameters

pDP	Pointer to the domain parameter value to be set.
tag	Tag specifying the desired domain parameter.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function assigns the value specified by *pDP* to a particular domain parameter of the DL-based cryptosystem. The domain parameter to be set up is determined by *tag* as follows:

- If *tag* == `IppDLPkeyP`, the function assigns value to the characteristic p, the size of the prime finite field GF(p).
- If *tag* == `IppDLPkeyR`, the function assigns value to the characteristic r, the prime divisor of (p-1) and the order of g.
- If *tag* == `IppDLPkeyG`, the function assigns value to the characteristic g, the element of GF(p) generating a multiplicative subgroup of order r.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsRangeErr`	Indicates an error condition if the Big Number specified by *pDP* is too big to be stored in the `IppsDLPState` context.

`ippStsBadArgErr` Indicates an error condition if some of the function parameters are invalid:

- Big Number specified by *pDP* is negative
- Domain parameter specified by *tag* does not match the `IppsDLPState` context.

DLPGetDP

Retrieves a particular domain parameter of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPGetDP(const IppsBigNumState* pDP, IppsDLPKeyTag tag,
    IppsDLPState* pCtx);
```

Parameters

pDP	Pointer to the output Big Number context.
tag	Tag specifying the domain parameter to be retrieved.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function retrieves value of a particular domain parameter of the DL-based cryptosystem from the `IppsDLPState` context and stores the value in the Big Number context **pDP*. The domain parameter to be retrieved is determined by *tag* as follows:

- If *tag* == `IppDLPkeyP`, the function retrieves value of the characteristic *p*, the size of the prime finite field GF(*p*).
- If *tag* == `IppDLPkeyR`, the function retrieves value of the characteristic *r*, the prime divisor of (*p*-1) and the order of *g*.
- If *tag* == `IppDLPkeyG`, the function retrieves value of the characteristic *g,* the element of GF(*p*) generating a multiplicative subgroup of order *r*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsOutOfRangeErr`	Indicates an error condition if the Big Number specified by *`pDP`* is too small for the DL parameter.
`ippStsBadArgErr`	Indicates an error condition if the domain parameter specified by the tag does not match the `IppsDLPState` context.

DLPGenKeyPair

Generates a private key and computes public keys of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPGenKeyPair(ippsBigNumState* pPrivate, ippsBigNumState*
    pPublic, IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

`pPrivate`	Pointer to the private key *privKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pCtx`	Pointer to the cryptosystem context.
`rndFunc`	Specifed Random Generator.
`pRndParam`	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates a private key *privKey* and computes a public key *pubKey* of the DL-based cryptosystem. The function employs specified *`rndFunc`* Random Generator to generate a pseudorandom private key. The value of the private key *privKey* is a random number that lies in the range of [2,*R*-2].

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsRangeErr`	Indicates an error condition if any of the Big Numbers specified by *`pPrivate`* and *`pPublic`* is too small for the DL key.

DLPPublicKey

Computes a public key from the given private key of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPPublicKey(const IppsBigNumState* pPrivate,
    IppsBigNumState* pPublic, IppsDLPState* pCtx);
```

Parameters

`pPrivate`	Pointer to the input private key *privKey*.
`pPublic`	Pointer to the output public key *pubKey*.
`pCtx`	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function computes a public key *pubKey* of the DL-based cryptosystem.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsInvalidPrivateKey`	Indicates an error condition if the *privKey* has an illegal value.
`ippStsRangeErr`	Indicates an error condition if Big Number specified by *`pPublic`* is too small for the DL public key.

DLPValidateKeyPair

Validates private and public keys of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPValidateKeyPair(const IppsBigNumState* pPrivate, const
    IppsBigNumState* pPublic, IppDLPResult* pResult, IppsDLPState* pCtx);
```

Parameters

`pPrivate`	Pointer to the input private key *privKey*.
`pPublic`	Pointer to the output public key *pubKey*.
`pResult`	Pointer to the validation result.
`pCtx`	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function validates the private key *privKey* and the public key *pubKey* of the DL-based cryptosystem. The result of the validation is stored in the `*pResult` and may be assigned to one of the enumerators listed below:

`ippDLValid`	Vvalidation has passed successfully.
`ippDLInvalidPrivateKey`	$1 < private < (R-1)$ is false.
`ippDLInvalidPublicKey`	$(1 < public \le (P-1)$ is false.
`ippDLInvalidKeyPair`	`public != G^private (mod P).`

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.

DLPSetKeyPair

Sets private and/or public keys of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPSetKeyPair(const IppsBigNumState* pPrivate, const
    IppsBigNumState* pPublic, IppsDLPState* pCtx);
```

Parameters

pPrivate	Pointer to the input private key *privKey*.
pPublic	Pointer to the output public key *pubKey*.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function stores the private key *priveKey* and public key *pubKey* in the cryptosystem context.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.

DLPGenerateDSA

Generates domain parameters of the DL-based cryptosystem over GF(p) to use DSA.

Syntax

```
ippStatus ippsDLPGenerateDSA(const IppsBigNumState* pSeedIn, int nTrials,
    IppsDLPState* pCtx, IppsBigNumState* pSeedOut, int* pCounter, IppBitSupplier
    rndFunc, void* pRndParam);
```

Parameters

pSeedIn	Pointer to the input *Seed*.
nTrials	Security parameter specified for the Miller-Rabin probable primality.
pCtx	Pointer to the cryptosystem context.
pSeedOut	Pointer to the output *Seed* value (if requested).
pCounter	Pointer to the *counter* value (if requested).
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates domain parameters of the DL-based cryptosystem over GF(*p*) to use DSA. The function uses a procedure specified in [FIPS PUB 186-2] for generating both a 160-bit randomized prime r and a *LpeBits* prime *p* based on the input *`*pSeedIn`*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsSizeErr`	Indicates an error condition if: *`peBits`* < 512, *`peBits`* is not divided by 64, *`reBits`* != 160.
`ippStsRangeErr`	Indicates an error condition if: bitsize of the input *Seed* value is less than 160, bitsize of the input *Seed* value is greater than *`peBits`*, not enough space to store the output *Seed* value (if requested).
`ippStsBadArgErr`	Indicates an error condition if *`nTrials`* < 1.
`ippStsInsuffucientEntropy`	Indicates a warning condition if prime generation fails due to a poor choice of the entropy.

DLPValidateDSA

Validates domain parameters of the DL-based cryptosystem over GF(p) to use DSA.

Syntax

```
IppStatus ippsDLPValidateDSA(int nTrials, IppDLResult* pResult,
    IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

nTrials	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the validation result.
pCtx	Pointer to the cryptosystem context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function validates domain parameters of the DL-based cryptosystem over GF(p) to use DSA. The result of validation is stored in the **pResult* and may be assigned to one of the enumerators listed below:

`ippDLValid`	Validation has passed successfully.
`ippDLBaseIsEven`	P is even.
`ippDLOrderIsEven`	R is even.
`ippDLInvalidBaseRange`	$P \le 2^{peBits-1}$ or $P \ge 2^{peBits}$.
`ippDLInvalidOrderRange`	$R \le 2^{reBits-1}$ or $R \ge 2^{reBits}$.
`ippDLCompositeBase`	P is not a prime.
`ippDLCompositeOrder`	R is not a prime.
`ippDLInvalidCofactor`	R is not divided by (P-1).
`ippDLInvalidGenerator`	$1 < G < (P\text{-}1)$ is false or G^R != 1 (mod P).

To ensure that both p and r are primes, the function applies *nTrial*-round Miller-Rabin primality test. Test data for primality test is provided by the specified *rndFunc* Random Generator.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

ippStsIncompleteContextErr	Indicates an error condition if the cryptosystem context has not been properly set up.
ippStsBadArgErr	Indicates an error condition if *nTrials* < 1.

DLPSignDSA

Performs the DSA digital signature signing operation.

Syntax

```
IppStatus ippsDLPSignDSA(const IppsBigNumState* pMsg, const
    IppsBigNumState* pPrivate, IppsBigNumState* pSignR, IppsBigNumState*
    pSignS, IppsDLPState* pCtx);
```

Parameters

pMsg	Pointer to the message representation *msgRep* to be signed.
pPrivate	Pointer to the signer's private key *privKey*.
pSignR	Pointer to the *r*-component of the signature.
pSignS	Pointer to the *s*-component of the signature.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function performs the DSA digital signature signing operation provided that the ephemeral signer's key pair (both private and public) was previously computed (generated by DLPGenKeyPair or computed by DLPPublicKey) and then set up into the DLP context by the DLPSetKeyPair function.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.

`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsMessageErr`	Indicates an error condition if the value of *msgRep* is greater than the multiplicative subgroup characteristic (q).
`ippStsInvalidPrivateKey`	Indicates an error condition if an illegal value has been assigned to *privKey*.
`ippStsRangeErr`	Indicates an error condition if any of the signature components has not enough space.

DLPVerifyDSA

Verifies the input DSA digital signature.

Syntax

```
IppStatus ippsDLPVerifyDSA(const IppsBigNumState* pMsg, const
    IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppDLResult*
    pResult, IppsDLPState* pCtx);
```

Parameters

pMsg	Pointer to the message representation *msgRep*.
pSignR	Pointer to the signature *r*-component to be verified.
pSignS	Pointer to the signature *s*-component to be verified.
pResult	Pointer to the result of the verification.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function verifies the input DSA digital signature's components `*pSignR` and `*pSignS` with the supplied message representation *msgRep*. Signer's public key must be stored by the `DLPSetKeyPair` function before the `DLPVerifyDSA` operation.

The function sets the `*pResult` to `ippDLValid` if it validates the input DSA digital signature, or to `ippDLInvalidSignature` if the DSA digital signature verification fails.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsMessageErr`	Indicates an error condition if the value of *msgRep* is greater than the multiplicative subgroup characteristic (q).

The example below illustrates the use of functions `DLPSignDSA` and `DLPVerifyDSA`. The example uses the `BigNumber` class and functions creating some cryptographic contexts, whose source code can be found in Appendix B.

Example 5-9 The Use of DLPSignDSA and DLPVerifyDSA

```
//
// known domain parameters
//
static const int M = 512; // DSA system bitsize
static const int L = 160; // DSA order  bitsize

static
BigNumber P("0x8DF2A494492276AA3D25759BB06869CBEAC0D83AFB8D0CF7" \
            "CBB8324F0D7882E5D0762FC5B7210EAFC2E9ADAC32AB7AAC" \
            "49693DFBF83724C2EC0736EE31C80291");

static
BigNumber Q("0xC773218C737EC8EE993B4F2DED30F48EDACE915F");
```

Example 5-9 The Use of DLPSignDSA and DLPVerifyDSA (continued)

```
static
BigNumber G("0x626D027839EA0A13413163A55B4CB500299D5522956CEFCB" \
            "3BFF10F399CE2C2E71CB9DE5FA24BABF58E5B79521925C9C" \
            "C42E9F6F464B088CC572AF53E6D78802");

//
// known DSA regular key pair
//
static
BigNumber X("0x2070B3223DBA372FDE1C0FFC7B2E3B498B260614");

static
BigNumber Y("0x19131871D75B1612A819F29D78D1B0D7346F7AA77BB62A85" \
            "9BFD6C5675DA9D212D3A36EF1672EF660B8C7C255CC0EC74" \
            "858FBA33F44C06699630A76B030EE333");


int DSAsign_verify_sample(void)
{
   // DLP context
   IppsDLPState *DLPState = newDLP(M, L);

   // set up DLP crypto system
   ippsDLPSet(P, Q, G, DLPState);

  // message
   Ipp8u message[] = "abc";

   // compute message digest to be signed
   Ipp8u md[SHA1_DIGEST_LENGTH/8];
   ippsSHA1MessageDigest(message, sizeof(message)-1, md);
```

Example 5-9 The Use of DLPSignDSA and DLPVerifyDSA (continued)

```
    BigNumber digest(0, BITS_2_WORDS(SHA1_DIGEST_LENGTH));
    ippsSetOctString_BN(md, SHA1_DIGEST_LENGTH/8, digest);

    // generate ephemeral key pair (ephX,ephY)
    BigNumber ephX(0, BITS_2_WORDS(L));
    BigNumber ephY(0, BITS_2_WORDS(M));

    IppsPRNGState* pRand = newPRNG();
    ippsDLPGenKeyPair(ephX, ephY, DLPState, ippsPRNGen, pRand);
    deletePRNG(pRand);
    //
    // generate signature
    //
    BigNumber signR(0, BITS_2_WORDS(L));      // R and S signature's component
    BigNumber signS(0, BITS_2_WORDS(L));
    ippsDLPSetKeyPair(ephX, ephY, DLPState);  // set up ephemeral keys
    ippsDLPSignDSA(digest, X,                 // sign digest
                   signR, signS,
                   DLPState);

    //
    // verify signature
    //
    ippsDLPSetKeyPair(0, Y, DLPState);        // set up regular public key
       IppDLResult result;
       ippsDLPVerifyDSA(digest, signR,signS,     // verify
                        &result, DLPState);

    deleteDLP(DLPState);
    return result==ippDLValid;
}
```

DLPGenerateDH

Generates domain parameters of the DL-based cryptosystem over GF(p) to use the DH Agreement scheme.

Syntax

```
IppStatus IppsDLPGenerateDH(const IppsBigNumState* pSeedIn, int nTrials,
    IppsDLPState* pCtx, IppsBigNumState* pSeedOut, int* pCounter,
    IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

pSeedIn	Pointer to the input *Seed*.
nTrials	Security parameter specified for the Miller-Rabin probable primality.
pCtx	Pointer to the cryptosystem context.
pSeedOut	Pointer to the output *Seed* value (if requested).
pCounter	Pointer to the *counter* value (if requested).
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates domain parameters of the DL-based cryptosystem over GF(p) to use Diffie-Hellman Agreement scheme. The function uses a procedure specified in [X9.42] for generating both randomized prime p and r based on the input **pSeedIn*.

Generated primes r and p are further validated through a *nTrial*-round Miller-Rabin primality test. Both generation and primality test procedures employ specified *rndFunc* Random Generator.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsSizeErr`	Indicates an error condition if: *peBits* < 512 or *reBits* < 160, *peBits* is not divided by 256.
`ippStsRangeErr`	Indicates an error condition if: bitsize of the input *Seed* value is less than *reBits*, not enough space to store the output *Seed* value (if requested).
`ippStsBadArgErr`	Indicates an error condition if *nTrials* < 1.
`ippStsInsuffucientEntropy`	Indicates a warning condition if prime generation fails due to a poor choice of the entropy.

DLPValidateDH

Validates domain parameters of the DL-based cryptosystem over GF(p) to use the DH Agreement scheme.

Syntax

```
IppStatus ippsDLPValidateDH(int nTrials, IppDLResult* pResult,
    IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

nTrials	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the validation result.
pCtx	Pointer to the cryptosystem context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function validates domain parameters of the DL-based cryptosystem over GF(p) to use Diffie-Hellman Agreement scheme. The result of validation is stored in the `*pResult` and may be assigned to one of the enumerators listed below:

`ippDLValid`	Validation has passed successfully.
`ippDLBaseIsEven`	P is even.
`ippDLOrderIsEven`	R is even.
`ippDLInvalidBaseRange`	$P \le 2^{peBits-1}$ or $P \ge 2^{peBits}$.
`ippDLInvalidOrderRange`	$R \le 2^{reBits-1}$ or $R \ge 2^{reBits}$.
`ippDLCompositeBase`	P is not a prime.
`ippDLCompositeOrder`	R is not a prime.
`ippDLInvalidCofactor`	R is not divided by (P-1).
`ippDLInvalidGenerator`	$1 < G < (P-1)$ is false or G^R != 1 (mod P).

To ensure that both p and r are primes, the function applies *nTrial*-round Miller-Rabin primality test. Test data for primality test is provided by the specified *rndFunc* Random Generator.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsBadArgErr`	Indicates an error condition if *nTrials* < 1.

DLPSharedSecretDH

Computes a shared field element by using the Diffie-Hellman scheme.

Syntax

```
IssStatus ippsDLPSharedSecretDH(const IppsBigNumState* pPrivateA, const
    IppsBigNumState* pPublicB, IppsBigNumState* pShare, IppsDLPState* pCtx);
```

Parameters

pPrivateA	Pointer to your own private key *privateKeyA*.
pPublicB	Pointer to the public key *pubKeyB* belonging to the other party.
pShare	Pointer to the shared secret element *Share*.
pCtx	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function computes a shared secret element FG(p) $pubKeyB^{privateKeyA} (mod\ p)$.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsRangeErr`	Indicates an error condition if *Share* does not have enough space.

Elliptic Curve Cryptography Functions

Intel® Integrated Performance Primitives (Intel® IPP) for cryptography offer functions allowing for different operations with an elliptic curve defined over a prime finite field GF(p) and binary finite field GF(2^m). The functions are based on standards ([IEEE P1363A], [SEC1], and [ANSI]. For more information on parameters recommended for the functions, see [SEC2].

The full list of Elliptic Curve Cryptography functions is given in Table 5-10.

Table 5-10 Intel IPP Elliptic Curve Cryptography Functions

Function Base Name	Operation
Functions Operating over GF(p)	
ECCPGetSize	Gets the size of the `IppsECCPState` context.
ECCPInit	Initializes context for the elliptic curve cryptosystem over GF(p).
ECCPSet	Sets up elliptic curve domain parameters over GF(p).
ECCPSetStd	Sets up a recommended set of elliptic curve domain parameters over GF(p).
ECCPGet	Retrieves elliptic curve domain parameters over GF(p).
ECCPGetOrderBitSize	Retrieves order size of the elliptic curve base point over GF(p) in bits
ECCPValidate	Checks validity of the elliptic curve domain parameters over GF(p).
ECCPPointGetSize	Gets the size of the `IppsECCPPoint` context in bytes for a point on the elliptic curve point defined over GF(p).
ECCPPointInit	Initializes context for a point on the elliptic curve defined over GF(p).
ECCPSetPoint	Sets coordinates of a point on the elliptic curve defined over GF(p).
ECCPSetPointAtInfinity	Sets the point at infinity.
ECCPGetPoint	Retrieves coordinates of the point on the elliptic curve defined over GF(p).
ECCPCheckPoint	Checks correctness of the point on the elliptic curve defined over GF(p).
ECCPComparePoint	Compares two points on the elliptic curve defined over GF(p).
ECCPNegativePoint	Finds an elliptic curve point which is an additive inverse for the given point over GF(p).
ECCPAddPoint	Computes the addition of two elliptic curve points over GF(p).
ECCPMulPointScalar	Performs scalar multiplication of a point on the elliptic curve defined over GF(p).
ECCPGenKeyPair	Generates a private key and computes public keys of the elliptic cryptosystem over GF(p).

Table 5-10 Intel IPP Elliptic Curve Cryptography Functions (continued)

Function Base Name	Operation
`ECCPPublicKey`	Computes a public key from the given private key of the elliptic cryptosystem over GF(*p*).
`ECCPValidateKeyPair`	Validates private and public keys of the elliptic cryptosystem over GF(*p*).
`ECCPSetKeyPair`	Sets private and/or public keys of the elliptic cryptosystem over GF(*p*).
`ECCPSharedSecretDH`	Computes a shared secret field element by using the Diffie-Hellman scheme.
`ECCPSharedSecretDHC`	Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.
`ECCPSignDSA`	Computes a digital signature over a message digest.
`ECCPVerifyDSA`	Verifies authenticity of the digital signature over a message digest (ECDSA).
`ECCPSignNR`	Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).
`ECCPVerifyNR`	Verifies authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).
Functions Operating over GF(2^m)	
`ECCBGetSize`	Gets the size of the `IppsECCBState` context.
`ECCBInit`	Initializes context for the elliptic curve cryptosystem over GF(2^m).
`ECCBSet`	Sets up elliptic curve domain parameters over GF(2^m).
`ECCBSetStd`	Sets up a recommended set of elliptic curve domain parameters over GF(2^m).
`ECCBGet`	Retrieves elliptic curve domain parameters over GF(2^m).
`ECCBGetOrderBitSize`	Retrieves order size of the elliptic curve base point over GF(2^m) in bits.
`ECCBValidate`	Checks validity of the elliptic curve domain parameters over GF(2^m).
`ECCBPointGetSize`	Gets the size of the IppsECCBPoint context in bytes for a point on the elliptic curve point defined over GF(2^m).
`ECCBPointInit`	Initializes context for a point on the elliptic curve defined over GF(2^m).
`ECCBSetPoint`	Sets coordinates of a point on the elliptic curve defined over GF(2^m).
`ECCBSetPointAtInfinity`	Sets the point at infinity.
`ECCBGetPoint`	Retrieves coordinates of the point on the elliptic curve defined over GF(2^m).
`ECCBCheckPoint`	Checks correctness of the point on the elliptic curve defined over GF(2^m).
`ECCBComparePoint`	Compares two points on the elliptic curve defined over GF(2^m).

Table 5-10 Intel IPP Elliptic Curve Cryptography Functions (continued)

Function Base Name	Operation
ECCBNegativePoint	Finds an elliptic curve point which is an additive inverse for the given point over GF(2^m).
ECCBAddPoint	Computes the addition of two elliptic curve points over GF(2^m).
ECCBMulPointScalar	Performs scalar multiplication of a point on the elliptic curve defined over GF(2^m).
ECCBGenKeyPair	Generates a private key and computes public keys of the elliptic cryptosystem over GF(2^m).
ECCBPublicKey	Computes a public key from the given private key of the elliptic cryptosystem over GF(2^m).
ECCBValidateKeyPair	Validates private and secret keys of the elliptic cryptosystem over GF(2^m).
ECCBSetKeyPair	Sets private and/or public keys in the elliptic cryptosystem over GF(2^m).
ECCBSharedSecretDH	Computes a shared secret field element by using the Diffie-Hellman scheme.
ECCBSharedSecretDHC	Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.
ECCBSignDSA	Computes a digital signature over a message digest.
ECCBVerifyDSA	Verifies authenticity of the digital signature over a message digest (ECDSA).
ECCBSignNR	Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).
ECCBVerifyNR	Computes authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Public key cryptography successfully allows to solve problems of information security by enabling secure communication over insecure channels. Although elliptic curves are well studied as a branch of mathematics, an interest to the cryptographic schemes based on elliptic curves is constantly rising due to the advantages that the elliptic curve algorithms provide in the wireless communications: shorter processing time and key length.

Elliptic curve cryptosystems (ECCs) implement a different way of creating public keys. Because elliptic curve calculation is based on the addition of the rational points in the (x,y) plane and it is difficult to solve a discrete logarithm from these points, a higher level of security is achieved through the cryptographic schemes that use the elliptic curves. The cryptographic systems that encrypt messages by using the properties of elliptic curves are hard to attack due to the extreme complexity of deciphering the private key.

Use of elliptic curves allows for shorter public key length and encourage cryptographers to create cryptosystems with the same or higher encryption strength as the RSA or DSA cryptosystems. Because of the relatively short key length, ECCs do encryption and decryption faster on the hardware that requires less computation processing volumes. For example, with a key length of 150-350 bits, ECCs provide the same encryption strength as the cryptosystems who have to use 600 -1400 bits. Alternatively, it requires considerably less time to create a digital signature with the ECDSA (Elliptic Curve Digital Signature Algorithm) algorithm rather than with the RSA algorithm even though you are using the same processor machine.

Functions Based on GF(*p*)

This section describes functions designed to specify the elliptic curve cryptosystem and perform various operations on the elliptic curve defined over a prime finite field. The examples of the operations are shown below:

- Setting up operations
  `ECCPSet` sets up elliptic curve domain parameters. `ECCPSetKeyPair` sets a pair of public and private keys for the given cryptosystem.
- Computation operations
  `ECCPAddPoint` adds two points on the elliptic curve. `ECCPMulPointScalar` performs the scalar multiplication of a point on the elliptic curve. `ECCPSignDSA` computes the digital signature of a message.
- Validation operations
  `ECCPValidate` checks validity of the elliptic curve domain parameters. `ECCPValidateKeyPair` validates correctness of the public and private keys.
- Generation operations
  `ECCPGenKeyPair` generates a private key and computes a public key for the given elliptic cryptosystem.
- Retrieval operations
  `ECCPGet` retrieves elliptic curve domain parameters. `ECCPGetOrderBitSize` retrieves the size of a base point in bytes.

All functions described in this section employ a context `IppsECCPState` that catches several auxiliary components specifying operations performed on the elliptic curve or entire elliptic cryptosystem. ECCP stands for Elliptic Curve Cryptography Prime and means that all functions whose name include this abbreviation perform operations over a prime finite field GF(*p*).

ECCPGetSize

Gets the size of the `IppsECCPState` *context.*

Syntax

```
IppStatus ippsECCPGetSize(int feBitSize, int *pSize);
```

Parameters

feBitSize	Size (in bits) of the field element.
pSize	Pointer to the size (in bytes) of the context.

Description

The function computes the size of the context in bytes for the elliptic cryptosystem over a prime finite field GF (p).

Context is a structure `IppsECCPState` designed to store information about the cryptosystem status.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 2.

ECCPInit

Initializes context for the elliptic curve cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPInit(int feBitSize, IppsECCPState* pECC);
```

Parameters

feBitSize	Size (in bits) of a field element.
pECC	Pointer to the cryptosystem context.

Description

The function initializes the context of the elliptic curve cryptosystem over the prime finite field GF(p).

Context is a structure `IppsECCPState` designed to store information about the cryptosystem status.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NIULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 2.

ECCPSet

Sets up elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippsECCPSet(const IppsBigNumState* pPrime, const
    IppsBigNumState* pA, const IppsBigNumState* pB, const
    IppsBigNumState* pGX, const IppsBigNumState* pGY, const
    IppsBigNumState* pOrder, int cofactor, IppsECCPState* pECC);
```

Parameters

pPrime	Pointer to the characteristic p of the prime finite field GF(p).
pA	Pointer to the coefficient A of the equation defining the elliptic curve.
pB	Pointer to the coefficient B of the equation defining the elliptic curve.
pGX	Pointer to the x-coordinate of the elliptic curve base point.
pGY	Pointer to the y-coordinate of the elliptic curve base point.

pOrder	Pointer to the order of the elliptic curve base point.
cofactor	Cofactor.
pECC	Pointer to the context of the cryptosystem.

Description

The function sets up the elliptic curve domain parameters over a prime finite field GF(p). These are as follows:

- *pPrime* sets up the characteristic p of a finite field GF(p) where p is a prime number.
- *pA*, *pB* set up the coefficients A and B of the equation defining the elliptic curve: $y^2 = x^3 + A \cdot x + B \pmod{p}$.
- *pGX*, *pGY* are pointers to the affine coordinates of the elliptic curve base point G.
- *pOrder* is a pointer to the order n of the elliptic curve base point G such that $n \cdot G = O$ where O is the point at infinity and n is a prime number.
- *cofactor* sets up the ratio h of a general number of points #E on the elliptic curve (including the point at infinity) to the order n of the base point : $h = \frac{\#E}{n}$.

The domain parameters are set in the cryptosystem context which must be already created by the `ippsECCPGetSize` and `ippsECCPInit` functions.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pPrime*, *pA*, *pB*, *pGX*, *pGY*, *pOrder*, and *pECC* is not valid.
`ippStsRangeErr`	Indicates an error condition if of one of the parameters pointed by *pPrime*, *pA*, *pB*, *pGX*, *pGY*, and *pOrder* cannot embed the *feBitSize* bits length or the value of `cofactor` is less than 1.

ECCPSetStd

Sets up a recommended set of elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippsECCPSetStd(IppECCType flag, IppsECCPState* pECC);
```

Parameters

flag	Set specifier.
pECC	Pointer to the cryptosystem context.

Description

The function sets a recommended set of elliptic curve domain parameters over a prime finite field GF(*p*).

The set is defined by the value of the parameter *flag*. Possible values of the parameter are as follows:

`IppECCPStd112r1`	For the cryptosystem context where *feBitSize*==112
`IppECCPStd112r2`	For the cryptosystem context where *feBitSize*==112
`IppECCPStd128r1`	For the cryptosystem context where *feBitSize*==128
`IppECCPStd128r2`	For the cryptosystem context where *feBitSize*==128
`IppECCPStd160r1`	For the cryptosystem context where *feBitSize*==160
`IppECCPStd160r2`	For the cryptosystem context where *feBitSize*==160
`IppECCPStd192r1`	For the cryptosystem context where *feBitSize*==192
`IppECCPStd224r1`	For the cryptosystem context where *feBitSize*==224
`IppECCPStd256r1`	For the cryptosystem context where *feBitSize*==256
`IppECCPStd384r1`	For the cryptosystem context where *feBitSize*==384
`IppECCPStd521r1`	For the cryptosystem context where *feBitSize*==521.

For more information on parameter values for the recommended elliptic curves, see [SEC2].

The cryptosystem context must be already created by the `ippsECCPGetSize` and `ippsECCPInit` functions. The value of *`feBitSize`* is applied when these functions are called and predetermines the possible choice of the *`flag`* value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the cryptosystem context is not valid.
`ippStsECCInvalidFlagErr`	Indicates an error condition if the value of the parameter *`flag`* is not valid.

ECCPGet

Retrieves elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippsECCPGet(IppsBigNumState* pPrime, IppsBigNumState* pA,
    IppsBigNumState* pB, IppsBigNumState* pGX,IppsBigNumState* pGY,
    IppsBigNumState* pOrder, int* cofactor,  IppsECCPState* pECC);
```

Parameters

`pPrime`	Pointer to the characteristic p of the prime finite field GF(p).
`pA`	Pointer to the coefficient A of the equation defining the elliptic curve.
`pB`	Pointer to the coefficient B of the equation defining the elliptic curve.
`pGX`	Pointer to the x-coordinate of the elliptic curve base point.
`pGY`	Pointer to the y-coordinate of the elliptic curve base point.
`pOrder`	Pointer to the order n of the elliptic curve base point.
`cofactor`	Pointer to the cofactor h.
`pECC`	Pointer to the context of the cryptosystem.

Description

The function retrieves elliptic curve domain parameters from the context of the elliptic cryptosystem over a finite field GF(*p*) and allocates them in accordance with the pointers `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `cofactor`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, or `pECC` is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of one of the parameters pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `pECC` is less than the value of `feBitSize` in the `ippsECCPInit` function.

ECCPGetOrderBitSize

Retrieves order size of the elliptic curve base point over GF(p) in bits.

Syntax

```
IppStatus ippsECCPGetOrderBitSize(int* pBitSize, IppsECCPState* pECC);
```

Parameters

pBitSize	Pointer to the size of the base point (in bits).
pECC	Pointer to the cryptosystem context.

Description

The function retrieves the order size (in bits) of the elliptic curve base point *G* from the context of elliptic cryptosystem over a prime finite field GF(*p*) and allocates it in accordance with the pointer `pBitsSize`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the cryptosystem context is not valid.

ECCPValidate

Checks validity of the elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippsECCPValidate(int nTrials, IppECResult* pResult,
    IppsECCPState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

`nTrials`	A number of attempts made to check the number for primality.
`pResult`	Pointer to the result received upon the check of the elliptic curve domain parameters.
`pECC`	Pointer to the cryptosystem context.
`rndFunc`	Specified Random Generator.
`pRndParam`	Pointer to Random Generator context.

Description

The function checks validity of the elliptic curve domain parameters over a prime finite field GF(p) and stores the result of the check in accordance with the pointer *`pResult`*.

Elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`. The purpose of the parameters *`rndFunc`*, *`pRndParam`*, and *`nTrials`* is analogous to that of the parameters *`rndFunc`*, *`pRndParam`*, and *`nTrials`* in the `ippsPrimeTest` function.

The result of the elliptic curve domain parameters check can take one of the following values:

`IppECIsValid`	The parameters are valid.

`IppECCompositeBase`	The prime finite field characterisitc p is a composite number.
`IppECIsNotAG`	The solutions of the elliptic curve equation do not form the abelian group because the only requirement that $4 \cdot a^3 + 27 \cdot b^2 \neq 0 \pmod{p}$ is not met.
`IppECPointIsNotValid`	The base point G is not on the elliptic curve.
`IppECCompositeOrder`	The order n of the base point G is a composite number.
`IppECInvalidOrder`	The order n of the base point G is not valid because the requirement that $n \cdot G = O$ where O is the point at infinity is not met.
`IppECIsWeakSSSA`	The order n of the base point G is equal to the finite field characteristic p.
`IppECIsWeakMOV`	The curve is excluded because it is subject to the MOV reduction attack.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`c`* or *`pECC`* is not valid.
`ippStsBadArgErr`	Indicates an error condition if the memory size of the parameter *`seed`* is less than five words (32 bytes in each) or the value of the parameter *`nTrails`* is less than 1.

ECCPPointGetSize

Gets the size of the `IppsECCPPoint` *context in bytes for a point on the elliptic curve point defined over GF(p).*

Syntax

```
IppStatus ippsECCPPointGetSize(int feBitSize, int* pSize);
```

Parameters

`feBitSize`	Size (in bits) of the field element.

pSize	Pointer to the context size.

Description

The function computes the context size in bytes for a point on the elliptic curve defined over a prime finite field GF(*p*).

Context is a structure `IppsECCPPoint` intended for storing the information about a point on the elliptic curve defined over GF(*p*).

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 2.

ECCPPointInit

Initializes the context for a point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPPointInit(int feBitSize, IppsECCPPoint* pPoint);
```

Parameters

feBitSize	Size (in bits) of the field element.
pECC	Pointer to the context of the elliptic curve point.

Description

The function initializes the context for a point on the elliptic curve defined over a finite field GF(*p*).

Context is a structure `IppsECCPPoint` intended for storing the information about a point on the elliptic curve defined over GF(*p*).

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *`feBitSize`* is less than 2.

ECCPSetPoint

Sets coordinates of a point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPSetPoint(const IppsBigNumState* pX, const
   IppsBigNumState* pY, IppsECCPPoint* pPoint, IppsECCPState* pECC);
```

Parameters

`pX`	Pointer to the *x*-coordinate of the point on the elliptic curve.
`pY`	Pointer to the *y*-coordinate of the point on the elliptic curve.
`pPoint`	Pointer to the context of the elliptic curve point.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function sets the coordinates of a point on the elliptic curve defined over a prime finite field GF(*p*).

The context of the point on the elliptic curve must be already created by functions: `ippsECCPPointGetSize` and `ippsECCPPointInit`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pX*, *pY*, *pPoint*, or *pECC* is not valid.

ECCPSetPointAtInfinity

Sets the point at infinity.

Syntax

```
IppStatus ippsECCPSetPointAtInfinity(IppsECCPPoint* pPoint,
    IppsECCPState* pECC);
```

Parameters

pPoint	Pointer to the context of the elliptic curve point.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function sets the point at infinity. The context of the elliptic curve point must be already created by functions: `ippsECCPPointGetSize` and `ippsECCPPointInit`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pPoint* or *pECC* is not valid.

ECCPGetPoint

Retrieves coordinates of the point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPGetPoint(IppsBigNumState* pX, IppsBigNumState* pY,
    const IppsECCPPoint* pPoint, IppsECCPState* pECC);
```

Parameters

pX	Pointer to the *x*-coordinate of the point on the elliptic curve.
pY	Pointer to the *y*-coordinate of the point on the elliptic curve.
pPoint	Pointer to the context of the elliptic curve point.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function retrieves the coordinates of the point on the elliptic curve defined over a prime finite field GF(*p*) from the point context and allocates them in accordance with the set pointers *pX* and *pY*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pX*, *pY*, *pPoint*, or *pECC* is not valid.

ECCPCheckPoint

Checks correctness of the point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPCheckPoint(const IppsECCPPoint* pP, IppECResult*
    pResult, IppsECCPState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point.
pResult	Pointer to the result of the check.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function checks the correctness of the point on the elliptic curve defined over a prime finite field GF(p) and allocates the result of the check in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

The result of the check for the correctness of the point can take one of the following values:

`IppECPointIsValid`	Point is on the elliptic curve.
`IppECPointIsNotValid`	Point is not on the elliptic curve and is not the point at infinity.
`IppECPointIsAtInfinite`	Point is the point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP* or *pECC* is not valid.

ECCPComparePoint

Compares two points on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPComparePoint(const IppsECCPPoint* pP, const
   IppsECCPPoint* pQ, IppECResult* pResult, IppsECCPState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pQ	Pointer to the elliptic curve point *Q*.
pResult	Pointer to the comparison result of two points: *P* and *Q*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function compares two points *P* and *Q* on the elliptic curve defined over a prime finite field GF(*p*) and allocates the comparison result in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

The comparison result of two points *P* and *Q* can take one of the following values:

`IppECPointIsEqual`	Points *P* and *Q* are equal.
`IppECPointIsNotEqual`	Points *P* and *Q* are different.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP* or *pECC* is not valid.

ECCPNegativePoint

Finds an elliptic curve point which is an additive inverse for the given point over GF(p).

Syntax

```
IppStatus ippsECCPNegativePoint(const IppsECCPPoint* pP, IppsECCPPoint*
    pR, IppsECCPState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function finds an elliptic curve point *R* over a prime finite field GF(*p*), which is an additive inverse of the given point *P*, that is, $P = -P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pR*, or *pECC* is not valid.

ECCPAddPoint

Computes the addition of two elliptic curve points over GF(p).

Syntax

```
IppStatus ippsECCPAddPoint(const IppsECCPPoint* pP, const IppsECCPPoint*
    pQ, IppsECCPPoint* pR, IppsECCPState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pQ	Pointer to the elliptic curve point *Q*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function calculates the addition of two elliptic curve points *P* and *Q* over a finite field GF(*p*) with the result in a point *R* such that $R = P + Q$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pQ*, *pR*, or *pECC* is not valid.

ECCPMulPointScalar

Performs scalar multiplication of a point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPMulPointScalar(const IppsECCPPoint* pP, const
    IppsBigNumState* pK, IppsECCPPoint* pR, IppsECCPState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pK	Pointer to the scalar *K*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function performs the *K* scalar multiplication of an elliptic curve point *P* over GF(*p*) with the result in a point *R* such that $R = K \cdot P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pK*, *pR*, or *pECC* is not valid.

ECCPGenKeyPair

Generates a private key and computes public keys of the elliptic cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPGenKeyPair(IppsBigNumState* pPrivate,
    IppsECCPPointState* pPublic, IppsECCPState* pECC, IppBitSupplier
    rndFunc, void* pRndParam);
```

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
pECC	Pointer to the context of the elliptic cryptosystem.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

The function generates a private key *privKey* and computes a public key *pubKey* of the elliptic cryptosystem over a finite field GF(*p*). The generation process employs the user specified *rndFunc* Random Generator.

The private key *privKey* is a number that lies in the range of [1, *n*-1] where *n* is the order of the elliptic curve base point.

The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$, where *G* is the base point of the elliptic curve.

The memory size of the parameter *privKey* pointed by *pPrivate* must be less than that of the base point which can also be defined by the function `ippsECCPGetOrderBitSize`.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions `ippsECCPPointGetSize` and `ippsECCPPointInit`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.
`ippStsSizeErr`	Indicates an error condition if the memory size of the parameter *privKey* pointed by *`pPrivate`* is less than that of the order of the elliptic curve base point.

ECCPPublicKey

Computes a public key from the given private key of the elliptic cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPPublicKey(const IppsBigNumState* pPrivate,
    IppsECCPPoint* pPublic, IppsECCPState* pECC);
```

Parameters

`pPrivate`	Pointer to the private key *privKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function computes the public key *pubKey* from the given private key *privKey* of the elliptic cryptosystem over a finite field GF(p).

The private key *privKey* is a number that lies in the range of [1, n-1] where n is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$, where G is the base point of the elliptic curve.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions `ippsECCPPointGetSize` and `ippsECCPPointInit`.

The elliptic curve domain parameters must be defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.
`ippStsIvalidPrivateKey`	Indicates an error condition if the value of the private key falls outside the range of [1, *n*-1].

ECCPValidateKeyPair

Validates private and public keys of the elliptic cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPValidateKeyPair(const IppsBigNumState* pPrivate, const
    IppsECCPPoint* pPublic, IppECResult* pResult, IppsECCPState* pECC);
```

Parameters

`pPrivate`	Pointer to the private key *privKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pResult`	Pointer to the validation result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function validates the private key *privKey* and public key *pubKey* of the elliptic cryptosystem over a finite field GF(p) and allocates the result of the validation in accordance with the pointer *`pResult`*.

The private key *privKey* is a number that lies in the range of [1, n-1], where n is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$ where G is the base point of the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

The result of the cryptosystem keys validation for correctness can take one of the following values:

`IppECValidKey`	Keys are valid.
`IppECInvalidKeyPair`	Keys are not valid because $privKey \cdot G \neq pubKey$
`IppECInvalidPrivateKey`	Key *privKey* falls outside the range of [1, *n*-1].
`IppECPointIsAtInfinite`	Key *pubKey* is the point at infinity.
`IppECInvalidPublicKey`	Key *pubKey* is not valid because $n \cdot pubKey \neq O$, where *O* is the point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pPrivate*, *pPublic*, or *pECC* is not valid.

ECCPSetKeyPair

Sets private and/or public keys of the elliptic cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPSetKeyPair(const IppsBigNumState* pPrivate, const
    IppsECCPPoint* pPublic, IppBool regular, IppsECCPState* pECC);
```

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
regular	Key status flag.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function sets a private key *privKey* and/or public key *pubKey* in the elliptic cryptosystem defined over a prime finite field GF(*p*).

The private key *privKey* is a number that lies in the range of [1, *n*-1], where *n* is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$, where *G* is the base point of the elliptic curve.

The two possible values of the parameter *`regular`* define the key timeliness status:

`ippTrue`	Keys are regular.
`ippFalse`	Keys are ephemeral.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.

ECCPSharedSecretDH

Computes a shared secret field element by using the Diffie-Hellman scheme.

Syntax

```
IppStatus ippsECCPSharedSecretDH(const IppsBigNumState* pPrivate, const
    IppsECCPPointState* pPublic, IppsBigNumState* pShare,
    IppsECCPState* pECC);
```

Parameters

`pPrivate`	Pointer to your own public key *pubKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pShare`	Pointer to the secret number *bnShare*.

pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function computes a secret number *bnShare*, which is a secret key shared between two participants of the cryptosystem.

In cryptography, metasyntactic names such as Alice as Bob are normally used as examples and in discussions and stand for participant A and participant B.

Both participants (Alice and Bob) use the cryptosystem for receiving a common secret point on the elliptic curve called a secret key. To receive a secret key, participants apply the Diffie-Hellman key-agreement scheme involving public key exchange. The value of the secret key entirely depends on participants.

According to the scheme, Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $pubKeyA = privKeyA \cdot G$ where *G* is a base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$ where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = privKeyA \cdot pubKeyB = privKeyA \cdot privKeyB \cdot G$
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = privKeyB \cdot pubKeyA = privKeyB \cdot privKeyA \cdot G$

Because the following equation is true $privKeyA \cdot privKeyB \cdot G = privKeyB \cdot privKeyA \cdot G$, the result of both calculations is the same, that is, the equation $shareA = shareB$ is true. The secret point serves as a secret key.

Shared secret *bnShare* is a x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPublic`*, *`pPShare`*, or *`pECC`* is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of *`bnShare`* pointed by *`pShare`* is less than the value of *`feBitSize`* in the function `ippsECCPInit`.
`ippStsShareKeyErr`	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.

ECCPSharedSecretDHC

Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.

Syntax

```
IppStatus ippsECCPSharedSecretDHC(const IppsBigNumState* pPrivate, const
    IppsECCPPointState* pPublic, IppsBigNumState* pShare,  IppsECCPState* pECC);
```

Parameters

`pPrivate`	Pointer to your own public key *pubKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pShare`	Pointer to the secret number *bnShare*.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function computes a secret number *`bnShare`* which is a secret key shared between two participants of the cryptosystem. Both participants (Alice and Bob) use the cryptosystem for getting a common secret point on the elliptic curve by using the Diffie-Hellman scheme and elliptic curve cofactor *h*.

Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $pubKeyA = privKeyA \cdot G$ where *G* is a base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$ where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = h \cdot privKeyA \cdot pubKeyB = h \cdot privKeyA \cdot privKeyB \cdot G$ where *h* is the elliptic curve cofactor.
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = h \cdot privKeyB \cdot pubKeyA = h \cdot privKeyB \cdot privKeyA \cdot G$ where *h* is the elliptic curve cofactor.

Shared secret *`bnShare`* is a x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPublic`*, *`pPShare`*, or *`pECC`* is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of *`bnShare`* pointed by *`pShare`* is less than the value of *`feBitSize`* in the function `ippsECCPInit`.
`ippStsShareKeyErr`	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.

ECCPSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippsECCPSignDSA(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pPrivate, IppsBigNumState* pSignX, IppsBigNumState*
    pSignY, IppsECCPState* pECC);
```

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted with a private key.
pPrivate	Pointer to the signer's regular private key.
pSignX	Pointer to the integer *r* of the digital signature.
pSignY	Pointer to the integer *s* of the digital signature.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

A message digest is a fixed size number derived from the original message with an applied hash function over the binary code of the message. The signer's private key and the message digest are used to create a signature.

A digital signature over a message consists of a pair of large numbers *r* and *s* which the given function computes.

The scheme used for computing a digital signature is analogue of the ECDSA scheme, an elliptic curve analogue of the DSA scheme. ECDSA assumes that the following keys are hitherto set by a message signer:

ephPrivKey	Ephemeral private key.
ephPubKey	Ephemeral public key.

The keys can be generated and set up by the unctions `ippsECCPGenKeyPair` and `ippsECCPSetKeyPair` with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pMsgDigest`*, *`pSignX`*, *`pSignY`*, or *`ECC`* is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by *`pMsgDigest`* falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base point *G*.
`ippStsRangeErr`	Indicates an error condition if one of the parameters pointed by *`pSignX`* or *`pSignY`* has a less memory size than the order *n* of the elliptic curve base point *G*.
`ippStsEphemeralKeyErr`	Indicates an error condition if the values of the ephemeral keys *ephPrivKey* and *ephPubKey* are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCPVerifyDSA

Verifies authenticity of the digital signature over a message digest (ECDSA).

Syntax

```
IppStatus ippsECCPVerifyDSA(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pSignX, const IppsBigNumState* pSignY,
    IppECResult* pResult, IppsECCPState* pECC);
```

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pSignX`	Pointer to the integer *r* of the digital signature.
`pSignY`	Pointer to the integer *s* of the digital signature.

`pResult`	Pointer to the digital signature verification result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function verifies authenticity of the digital signature over a message digest *msg*. The signature consists of two large integers: *r* and *s*.

The scheme used to verify the signature is an elliptic curve analogue of the DSA scheme and assumes that the following cryptosystem key be hitherto set:

regPubKey	Message sender's regular public key.

The *regPubKey* is set by the function `ippsECCPSetKeyPair`.

The result of the digital signature verification can take one of two possible values:

`IppsECSignIsValid`	Digital signature is valid.
`IppsECSignIsInvalid`	Digital signature is not valid.

The call to the `ippsECCPVerifyDSA` function must be preceded by the call to the `ippsECCPSignDSA` function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pMsgDigest`, `pSignX`, `pSignY`, or `ECC` is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by `pMsgDigest` falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base base point *G*.

ECCPSignNR

Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsECCPSignNR(const IppsBigNumState* pMsgDigest,
   IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState*
   pECC);
```

Parameters

pMsgDigest	Pointer to the message digest *msg*.
pSignX	Pointer to the integer *r* of the digital signature.
pSignY	Pointer to the integer *s* of the digital signature.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function computes two large numbers *r* and *s* which form the digital signature over a message digest *msg*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys are hitherto set up by the message sender:

regPrivKey	Regular private key.
ephPrivKey	Ephemeral private key.
ephPubKey	Ephemeral public key.

The keys can be generated and set up by the functions `ippsECCPGenKeyPair` and `ippsECCPSetKeyPair` with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pMsgDigest`, `pSignX`, `pSignY`, or `ECC` is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by `pMsgDigest` falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base point *G*.
`ippStsRangeErr`	Indicates an error condition if one of the parameters pointed by `pSignX` or `pSignY` has a less memory size than the order *n* of the elliptic curve base point *G*.
`ippStsEphemeralKeyErr`	Indicates an error condition if the values of the ephemeral keys *ephPrivKey* and *ephPubKey* are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCPVerifyNR

Verifies authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsECCPVerifyNR(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pSignX, const IppsBigNumState* pSignY,
    IppECResult* pResult, IppsECCPState* pECC);
```

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pSignX`	Pointer to the integer *r* of the digital signature.
`pSignY`	Pointer to the integer *s* of the digital signature.
`pResult`	Pointer to the digital signature verification result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function verifies authenticity of the digital signature over a message digest *msg*. The signature is presented with two large integers *r* and *s*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys be hitherto set up by the message sender:

regPubKey	Message sender's regular private key.

The key can be generated and set up by the function `ippsECCPGenKeyPair`.

The result of the digital signature verification can take one of two possible values:

`IppsECSignIsValid`	The digital signature is valid.
`IppsECSignIsInvalid`	The digital signature is not valid.

The call to the `ippsECCPVerifyNR` function must be preceded by the call to the `ippsECCPSignNR` function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCPSet` or `ippsECCPSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pMsgDigest`, `pSignX`, `pSignY`, or `ECC` is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by `pMsgDigest` falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base point *G*.

Functions Based on GF(2^m)

This section describes functions designed to specify the elliptic curve cryptosystem and perform various operations on the elliptic curve defined over a binary finite field. The examples of the operations are illustrated below:

- Setting up operations
  `ECCBSet` sets up elliptic curve domain parameters. `ECCBSetKeyPair` sets a pair of public and private keys for the given cryptosystem.
- Computation operations
  `ECCBAddPoint` adds two points on the elliptic curve. `ECCBMulPointScalar` performs the scalar multiplication of a point on the elliptic curve. `ECCBSignDSA` computes the digital signature of a message.
- Validation operations
  `ECCBValidate` checks validity of the elliptic curve domain parameters. `ECCBValidateKeyPair` validates correctness of the public and private keys.
- Generation operations
  `ECCBGenKeyPair` generates a private key and computes a public key for the given elliptic cryptosystem.
- Retrieval operations
  `ECCBGet` retrieves elliptic curve domain parameters. `ECCBGetOrderBitSize` retrieves the size of a base point in bytes.

All functions described in this section employ a context `IppsECCBState` that catches several auxiliary components specifying operations performed on the elliptic curve or entire elliptic cryptosystem. ECCB stands for Elliptic Curve Cryptography Binary and means that all functions whose name include this abbreviation perform operations over a binary finite field GF(2m).

ECCBGetSize

Gets the size of the `IppsECCBState` *context.*

Syntax

```
IppStatus ippsECCBGetSize(int feBitSize, int *pSize);
```

Parameters

feBitSize	Size (in bits) of the field element.
pSize	Pointer to the size of the context (in bytes).

Description

The function computes the size of the context in bytes for the elliptic cryptosystem over a binary finite field GF (2^m).

Context is a structure `IppsECCBState` designed to store information about the cryptosystem status.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 1.

ECCBInit

Initializes context for the elliptic curve cryptosystem over GF(2^m).

Syntax

```
IppStatus ippsECCBInit(int feBitSize, IppsECCBState* pECC)
```

Parameters

feBitSize	Size (in bits) of a field element.
pECC	Pointer to the cryptosystem context.

Description

The function initializes the context of the elliptic curve cryptosystem over a binary finite field GF (2^m).

Context is a structure `IppsECCBState` designed to store information about the cryptosystem status.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *`feBitSize`* is less than 1.

ECCBSet

Sets up elliptic curve domain parameters over GF(2^m).

Syntax

```
IppStatus ippsECCBSet(const IppsBigNumState* pPrime, const
    IppsBigNumState* pA, const IppsBigNumState* pB, const
    IppsBigNumState* pGX, const IppsBigNumState* pGY, const
    IppsBigNumState* pOrder, int cofactor, IppsECCBState* pECC);
```

Parameters

pPrime	Pointer to the irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field GF(2^m).
pA	Pointer to the coefficient A of the equation defining the elliptic curve.
pB	Pointer to the coefficient B of the equation defining the elliptic curve.
pGX	Pointer to the x-coordinate of the elliptic curve base point.
pGY	Pointer to the y-coordinate of the elliptic curve base point.
pOrder	Pointer to the order of the elliptic curve base point.
cofactor	Cofactor.
pECC	Pointer to the context of the cryptosystem.

Description

The function sets up the elliptic curve domain parameters over a binary finite field GF(2^m). These are as follows:

- `pPrime` sets up the the irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field GF(2^m).
- `pA`, `pB` set up the coefficients A and B of the equation defining the elliptic curve: $y^2 + x \cdot y = x^3 + A \cdot x^2 + B$ in GF(2^m).
- `pGX`, `pGY` set up coordinates of the elliptic curve base point G.
- `pOrder` sets up the order n of the elliptic curve base point G such that $n \cdot G = O$ where O is the point at infinity and n is a prime number.
- `cofactor` sets up the ratio h of a general number of points #E on the elliptic curve (including the point at infinity) to the order n of the base point : $h = \frac{\#E}{n}$.

The domain parameters are set in the cryptosystem context which must be already created by the `ippsECCBGetSize` and `ippsECCBInit` functions.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `pECC` is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of one of the parameters pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `pECC` is more than the value of `feBitSize` in the `ippsECCBInit` function or the value of `cofactor` is less than or equal to zero.

ECCBSetStd

Sets up a recommended set of elliptic curve domain parameters over GF(2^m).

Syntax

```
IppStatus ippsECCBSetStd(IppECCType flag, IppsECCBState* pECC);
```

Parameters

flag	Set specifier.
pECC	Pointer to the cryptosystem context.

Description

The function sets a recommended set of elliptic curve domain parameters over a binary finite field GF(2^m).

The set is defined by the value of the parameter *flag*. Possible values of the parameter are as follows:

`IppECCBStd113r1`	For the cryptosystem context where *feBitSize*==113
`IppECCBStd113r2`	For the cryptosystem context where *feBitSize*==113
`IppECCBStd131r1`	For the cryptosystem context where *feBitSize*==131
`IppECCBStd131r2`	For the cryptosystem context where *feBitSize*==131
`IppECCBStd163k1`	For the cryptosystem context where *feBitSize*==163
`IppECCBStd163r1`	For the cryptosystem context where *feBitSize*==163
`IppECCBStd163r2`	For the cryptosystem context where *feBitSize*==163
`IppECCBStd193r1`	For the cryptosystem context where *feBitSize*==193
`IppECCBStd193r2`	For the cryptosystem context where *feBitSize*==193
`IppECCBStd233k1`	For the cryptosystem context where *feBitSize*==233
`IppECCBStd233r1`	For the cryptosystem context where *feBitSize*==233
`IppECCBStd239k1`	For the cryptosystem context where *feBitSize*==239
`IppECCBStd283k1`	For the cryptosystem context where *feBitSize*==283
`IppECCBStd283r1`	For the cryptosystem context where *feBitSize*==283
`IppECCBStd409k1`	For the cryptosystem context where *feBitSize*==409.
`IppECCBStd409r1`	For the cryptosystem context where *feBitSize*==409.
`IppECCBStd571k1`	For the cryptosystem context where *feBitSize*==571.
`IppECCBStd571r1`	For the cryptosystem context where *feBitSize*==571.

For more information on parameter values for the recommended elliptic curves, see [SEC2].

The cryptosystem context must be already created by the `ippsECCBGetSize` and `ippsECCBInit` functions. The value of *`feBitSize`* is applied when these function are called and predetermines the possible choice of the *`flag`* value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the cryptosystem context is not valid.
`ippStsECCInvalidFlagErr`	Indicates an error condition if the value of the parameter *`flag`* is not valid.

ECCBGet

Retrieves elliptic curve domain parameters over $GF(2^m)$.

Syntax

```
IppStatus ippsECCBGet(IppsBigNumState* pPrime, IppsBigNumState* pA,
    IppsBigNumState* pB, IppsBigNumState* pGX,IppsBigNumState* pGY,
    IppsBigNumState* pOrder, int* cofactor,  IppsECCBState* pECC);
```

Parameters

`pPrime`	Pointer to the irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field $GF(2^m)$.
`pA`	Pointer to the coefficient A of the equation defining the elliptic curve.
`pB`	Pointer to the coefficient B of the equation defining the elliptic curve.
`pGX`	Pointer to the x-coordinate of the elliptic curve base point.
`pGY`	Pointer to the y-coordinate of the elliptic curve base point.
`pOrder`	Pointer to the order n of the elliptic curve base point.
`cofactor`	Pointer to the cofactor h.
`pECC`	Pointer to the context of the cryptosystem.

Description

The function retrieves elliptic curve domain parameters from the context of the elliptic cryptosystem over a binary finite field GF(2^m) and allocates them in accordance with the pointers `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `cofactor`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, or `pECC` is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of one of the parameters pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `pECC` is less than the value of `feBitSize` in the `ippsECCBInit` function.

ECCBGetOrderBitSize

Retrieves order size of the elliptic curve base point over GF(2^m) in bits.

Syntax

```
IppStatus ippsECCBGetOrderBitSize(int* pBitSize, IppsECCBState* pECC);
```

Parameters

`pBitSize`	Pointer to the size of the base point (in bits).
`pECC`	Pointer to the cryptosystem context.

Description

The function retrieves the order size (in bits) of the elliptic curve base point *G* from the context of elliptic cryptosystem over a binary finite field GF(2^m) and allocates it in accordance with the pointer `pBitsSize`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if the cryptosystem context is not valid.

ECCBValidate

Checks validity of the elliptic curve domain parameters over GF(2^m).

Syntax

```
IppStatus ippsECCBValidate(int nTrials, IppECResult* pResult,
    IppsECCBState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

nTrials	A number of attempts made to check the number for primality.
pResult	Pointer to the result received upon the check of the elliptic curve domain parameters.
pECC	Pointer to the cryptosystem context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

The function checks validity of the elliptic curve domain parameters over a binary finite field GF(2^m) and stores the result of the check in accordance with the pointer *pResult*.

Elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`. The purpose of the parameters *rndFunc*, *pRndParam*, and *nTrials* is analogous to that of the parameters *rndFunc*, *pRndParam*, and *nTrials* in the `ippsPrimeTest` function.

The result of the elliptic curve domain parameters check can take one of the following values:

`IppECIsValid`	The parameters are valid.
`IppECComplicatedBase`	The irreducible binary polynomial `f(x)` of degree *m* which specifies the presentation of the field GF(2^m) is not valid because the set of polynomials consists of more than five elements.
`IppECCompositeBase`	The binary polynomial `f(x)` is not irreducible.
`IppECIsSupersingular`	The coefficient in the elliptic curve equation is `NULL`.
`IppECPointAtInfinite`	The elliptic curve base point G is the point at infinity.
`IppECPointIsNotValid`	Base point *G* is not on the elliptic curve.
`IppECCompositeOrder`	The order *n* of the base point *G* is a composite number.
`IppECInvalidOrder`	The order *n* of the base point *G* is not valid because the requirement that $n \cdot G = O$ where *O* is the point at infinity is not met.
`IppECIsWeakSSSA`	$h \cdot n = 2^m$ where *h* is a cofactor and *n* is the order *n* of the base point.
`IppECIsWeakMOV`	The curve is excluded because it is subject to the MOV reduction attack.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`c`* or *`pECC`* is not valid.
`ippStsBadArgErr`	Indicates an error condition if the memory size of the parameter *`seed`* is less than five words (32 bytes in each) or the value of the parameter *`nTrails`* is less than 1.

ECCBPointGetSize

Gets the size of the `IppsECCBPoint` *context in bytes for a point on the elliptic curve point defined over GF(*2^m*).*

Syntax

```
IppStatus ippsECCBPointGetSize(int feBitSize, int* pSize);
```

Parameters

feBitSize	Size (in bits) of the field element.
pSize	Pointer to the context size.

Description

The function computes the context size in bytes for a point on the elliptic curve defined over a binary finite field GF(2^m).

Context is a structure `IppsECCBPoint` intended for storing the information about a point on the elliptic curve defined over GF(2^m).

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 1.

ECCBPointInit

Initializes the context for a point on the elliptic curve defined over GF(2^m*).*

Syntax

```
IppStatus ippsECCBPointInit(int feBitSize, IppsECCBPoint* pPoint);
```

Parameters

feBitSize	Size (in bits) of the field element.
pECC	Pointer to the context of the elliptic curve point.

Description

The function initializes the context for a point on the elliptic curve defined over a binary finite field GF(2^m).

Context is a structure `IppsECCBPoint` intended for storing the information about a point on the elliptic curve defined over GF(2^m).

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsSizeErr`	Indicates an error condition if the value of the parameter *feBitSize* is less than 1.

ECCBSetPoint

Sets coordinates of a point on the elliptic curve defined over GF(2^m).

Syntax

```
IppStatus ippsECCBSetPoint(const IppsBigNumState* pX, const
    IppsBigNumState* pY, IppsECCBPoint* pPoint, IppsECCBState* pECC);
```

Parameters

pX	Pointer to the *x*-coordinate of the point on the elliptic curve.
pY	Pointer to the *y*-coordinate of the point on the elliptic curve.
pPoint	Pointer to the context of the elliptic curve point.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function sets the coordinates of a point on the elliptic curve defined over a binary finite field GF(2^m).

The context of the point on the elliptic curve must be already created by functions: `ippsECCBPointGetSize` and `ippsECCBPointInit`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pX*, *pY*, *pPoint*, or *pECC* is not valid.

ECCBSetPointAtInfinity

Sets the point at infinity.

Syntax

```
IppStatus ippsECCBSetPointAtInfinity(IppsECCBPoint* pPoint,
    IppsECCBState* pECC);
```

Parameters

pPoint	Pointer to the context of the elliptic curve point.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function sets the point at infinity. The context of the elliptic curve point must be already created by functions: `ippsECCBPointGetSize` and `ippsECCBPointInit`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPoint`* or *`pECC`* is not valid.

ECCBGetPoint

Retrieves coordinates of the point on the elliptic curve defined over GF(2^m).

Syntax

```
IppStatus ippsECCBGetPoint(IppsBigNumState* pX, IppsBigNumState* pY,
    const IppsECCBPoint* pPoint, IppsECCBState* pECC);
```

Parameters

`pX`	Pointer to the *x*-coordinate of the point on the elliptic curve.
`pY`	Pointer to the *y*-coordinate of the point on the elliptic curve.
`pPoint`	Pointer to the context of the elliptic curve point.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function retrieves the coordinates of the point on the elliptic curve defined over a binary finite field GF(2^m) from the point context and allocates them in accordance with the set pointers *`pX`* and *`pY`*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pX`*, *`pY`*, *`pPoint`*, or *`pECC`* is not valid.

ECCBCheckPoint

Checks correctness of the point on the elliptic curve defined over GF(2^m).

Syntax

```
IppStatus ippsECCBCheckPoint(const IppsECCBPoint* pP, IppECResult*
    pResult, IppsECCBState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point.
pResult	Pointer to the result of the check.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function checks the correctness of the point on the elliptic curve defined over a binary finite field GF(2^m) and allocates the result of the check in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

The result of the check for the correctness of the point can take one of the following values:

`IppECPointIsValid`	Point is on the elliptic curve.
`IppECPointIsNotValid`	Point is not on the elliptic curve and is not the point at infinity.
`IppECPointIsAtInfinite`	Point is the point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP* or *pECC* is not valid.

ECCBComparePoint

Compares two points on the elliptic curve defined over GF(2^m).

Syntax

```
IppStatus ippsECCBComparePoint(const IppsECCBPoint* pP, const
    IppsECCBPoint* pQ, IppECResult* pResult, IppsECCBState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pQ	Pointer to the elliptic curve point *Q*.
pResult	Pointer to the comparison result of two points: *P* and *Q*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function compares two points *P* and *Q* on the elliptic curve defined over a binary finite field GF(2^m) and allocates the comparison result in accordance with the pointer `pResult`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

The comparison result of two points *P* and *Q* can take one of the following values:

`IppECPointIsEqual`	Points *P* and *Q* are equal.
`IppECPointIsNotEqual`	Points *P* and *Q* are different.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP* or *pECC* is not valid.

ECCBNegativePoint

Finds the elliptic curve point which is an additive inverse for the given point over GF(2^m).

Syntax

```
IppStatus ippsECCBNegativePoint(const IppsECCBPoint* pP, IppsECCBPoint*
    pR, IppsECCBState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function finds an elliptic curve point *R* over a binary finite field GF(2^m) which is an additive inverse of the given point *P*, i.e., $P = -P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pR*, or *pECC* is not valid.

ECCBAddPoint

Computes the addition of two elliptic curve points over $GF(2^m)$.

Syntax

```
IppStatus ippsECCBAddPoint(const IppsECCBPoint* pP, const IppsECCBPoint*
    pQ, IppsECCBPoint* pR, IppsECCBState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pQ	Pointer to the elliptic curve point *Q*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function calculates the addition of two elliptic curve points *P* and *Q* over a binary finite field $GF(2^m)$ with the result in a point *R* such that $R = P + Q$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pQ*, *pR*, or *pECC* is not valid.

ECCBMulPointScalar

Performs scalar multiplication of a point on the elliptic curve defined over GF(2^m).

Syntax

```
IppStatus ippsECCBMulPointScalar(const IppsECCBPoint* pP, const
    IppsBigNumState* pK, IppsECCBPoint* pR, IppsECCBState* pECC);
```

Parameters

pP	Pointer to the elliptic curve point *P*.
pK	Pointer to the scalar *K*.
pR	Pointer to the elliptic curve point *R*.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function performs the *K* scalar multiplication of an elliptic curve point *P* over GF(2^m) with the result in a point *R* such that $R = K \cdot P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pP*, *pK*, *pR*, or *pECC* is not valid.

ECCBGenKeyPair

Generates a private key and computes public keys of the elliptic cryptosystem over GF(2^m).

Syntax

```
IppStatus ippsECCBGenKeyPair(IppsBigNumState* pPrivate,
    IppsECCBPointState* pPublic, IppsECCBState* pECC, IppBitSupplier
    rndFunc, void* pRndParam);
```

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
pECC	Pointer to the context of the elliptic cryptosystem.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

The function generates a private key *privKey* and computes a public key *pubKey* of the elliptic cryptosystem over a binary finite field GF(2^m). The generation process employs user specified *rndFunc* Random Generator.

The private key *privKey* is a number that lies in the range of [1, n-1] where n is the order of the elliptic curve base point.

The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$ where G is the base point of the elliptic curve.

The memory size of the parameter *privKey* pointed by *pPrivate* must be less than that of the base point which can also be defined by the function `ippsECCBGetOrderBitSize`.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions `ippsECCBPointGetSize` and `ippsECCBPointInit`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.
`ippStsSizeErr`	Indicates an error condition if the memory size of the parameter *privKey* pointed by *`pPrivate`* is less than that of the order of the elliptic curve base point.

ECCBPublicKey

Computes a public key from the given private key of the elliptic cryptosystem over GF(2^m).

Syntax

```
IppStatus ippsECCBPublicKey(const IppsBigNumState* pPrivate,
    IppsECCBPoint* pPublic, IppsECCBState* pECC);
```

Parameters

`pPrivate`	Pointer to the private key *privKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function computes the public key *pubKey* from the given private key *privKey* of the elliptic cryptosystem over a binary finite field GF(2^m).

The private key *privKey* is a number that lies in the range of [1, *n*-1] where *n* is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$ where *G* is the base point of the elliptic curve.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions `ippsECCBPointGetSize` and `ippsECCBPointInit`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.
`ippStsIvalidPrivateKey`	Indicates an error condition if the value of the private key falls outside the range of [1, *n*-1].

ECCBValidateKeyPair

Validates private and secret keys of the elliptic cryptosystem over GF(2^m).

Syntax

```
IppStatus ippsECCBValidateKeyPair(const IppsBigNumState* pPrivate, const
    IppsECCBPoint* pPublic, IppECResult* pResult, IppsECCBState* pECC);
```

Parameters

`pPrivate`	Pointer to the private key *privKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pResult`	Pointer to the validation result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function validates the private key *privKey* and public key *pubKey* of the elliptic cryptosystem over a binary finite field GF(2^m) and allocates the result of the validation in accordance with the pointer *`pResult`*.

The private key *privKey* is a number that lies in the range of [1, *n*-1] where *n* is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$ where *G* is the base point of the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

The result of the cryptosystem keys validation for correctness can take one of the following values:

`IppECValidKey`	Keys are valid.
`IppECInvalidKeyPair`	Keys are not valid because $privKey \cdot G \neq pubKey$
`IppECInvalidPrivateKey`	Key *privKey* falls outside the range of [1, *n*-1].
`IppECPointIsAtInfinite`	Key *pubKey* is the point at infinity.
`IppECInvalidPublicKey`	Key *pubKey* is not valid because $n \cdot pubKey \neq O$ where *O* is the point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pPrivate*, *pPublic*, or *pECC* is not valid.

ECCBSetKeyPair

Sets private and/or public keys in the elliptic cryptosystem over GF(2^m).

Syntax

```
IppStatus ippsECCBSetKeyPair(const IppsBigNumState* pPrivate, const
    IppsECCBPoint* pPublic, IppBool regular, IppsECCBState* pECC);
```

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
regular	Key status flag.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function sets the private key *privKey* and/or public key *pubKey* in the elliptic cryptosystem defined over a binary finite field GF(2^m).

The private key *privKey* is a number that lies in the range of [1, *n*-1] where *n* is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$ where *G* is the base point of the elliptic curve.

The two possible values of the parameter *`regular`* define the key timeliness status:

`ippTrue`	Keys are regular.
`ippFalse`	Keys are ephemeral.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPrivate`*, *`pPublic`*, or *`pECC`* is not valid.

ECCBSharedSecretDH

Computes a shared secret field element by using the Diffie-Hellman scheme.

Syntax

```
IppStatus ippsECCBSharedSecretDH(const IppsBigNumState* pPrivate, const
    IppsECCBPointState* pPublic, IppsBigNumState* pShare, IppsECCBState* pECC);
```

Parameters

`pPrivate`	Pointer to your own public key *pubKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pShare`	Pointer to the secret number *bnShare*.

`pECC` Pointer to the context of the elliptic cryptosystem.

Description

The function computes a secret number `bnShare`, which is a secret key shared between two participants of the cryptosystem.

In cryptography, metasyntactic names such as Alice as Bob are normally used as examples and in discussions and stand for participant A and participant B.

Both participants (Alice and Bob) use the cryptosystem for receiving a common secret point on the elliptic curve called a secret key. To receive a secret key, participants apply the Diffie-Hellman key-agreement scheme involving public key exchange. The value of the secret key entirely depends on participants.

According to the scheme, Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $pubKeyA = privKeyA \cdot G$ where *G* is a base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$ where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = privKeyA \cdot pubKeyB = privKeyA \cdot privKeyB \cdot G$
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = privKeyB \cdot pubKeyA = privKeyB \cdot privKeyA \cdot G$

Because the following equation is true $privKeyA \cdot privKeyB \cdot G = privKeyB \cdot privKeyA \cdot G$, the result of both calculations is the same, that is, the equation $shareA = shareB$ is true. The secret point serves as a secret key.

Shared secret `bnShare` is a x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr` Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pPublic`*, *`pPShare`*, or *`pECC`* is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of *`bnShare`* pointed by *`pShare`* is less than the value of *`feBitSize`* in the function `ippsECCBInit`.
`ippStsShareKeyErr`	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.

ECCBSharedSecretDHC

Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.

Syntax

```
IppStatus ippsECCBSharedSecretDHC(const IppsBigNumState* pPrivate, const
    IppsECCBPointState* pPublic, IppsBigNumState* pShare,  IppsECCBState* pECC);
```

Parameters

`pPrivate`	Pointer to your own public key *pubKey*.
`pPublic`	Pointer to the public key *pubKey*.
`pShare`	Pointer to the secret number *bnShare*.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function computes a secret number *`bnShare`* which is a secret key shared between two participants of the cryptosystem. Both participants (Alice and Bob) use the cryptosystem for getting a common secret point on the elliptic curve by using the Diffie-Hellman scheme and elliptic curve cofactor *h*.

Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $pubKeyA = privKeyA \cdot G$ where *G* is a base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$ where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = h \cdot privKeyA \cdot pubKeyB = h \cdot privKeyA \cdot privKeyB \cdot G$ where *h* is the elliptic curve cofactor.
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = h \cdot privKeyB \cdot pubKeyA = h \cdot privKeyB \cdot privKeyA \cdot G$ where *h* is the elliptic curve cofactor.

Shared secret *bnShare* is a x-coordinate of the secret point on the elliptic curve.

To define a secret key, the call to the `ippsECCBSharedSecretDH` function must be preceded by the call to the `ippsECCBSetKeyPair` function.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *pPublic*, *pPShare*, or *pECC* is not valid.
`ippStsRangeErr`	Indicates an error condition if the memory size of *bnShare* pointed by *pShare* is less than the value of *feBitSize* in the function `ippsECCBInit`.
`ippStsShareKeyErr`	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.

ECCBSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippsECCBSignDSA(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pPrivate, IppsBigNumState* pSignX, IppsBigNumState*
    pSignY, IppsECCBState* pECC);
```

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted with a private key.
pPrivate	Pointer to the signer's regular private key.
pSignX	Pointer to the integer *r* of the digital signature.
pSignY	Pointer to the integer *s* of the digital signature.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

A message digest is a fixed size number derived from the original message with an applied hash function over the binary code of the message. The signer's private key and the message digest are used to create a signature.

A digital signature over a message consists of a pair of large numbers *r* and *s* which the given function computes.

The scheme used for computing a digital signature is analogue of the ECDSA scheme, an elliptic curve analogue of the DSA scheme. ECDSA assumes that the following keys are hitherto set by a message signer:

ephPrivKey	Ephemeral private key.
ephPubKey	Ephemeral public key.

The keys can be generated and set up by the functions `ippsECCBGenKeyPair` and `ippsECCBSetKeyPair` with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pMsgDigest`*, *`pSignX`*, *`pSignY`*, or *`ECC`* is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by *`pMsgDigest`* falls outside the range of [1, 1-n] where n is the order of the elliptic curve base point G.
`ippStsRangeErr`	Indicates an error condition if one of the parameters pointed by *`pSignX`* or *`pSignY`* has a less memory size than the order n of the elliptic curve base point G.
`ippStsEphemeralKeyErr`	Indicates an error condition if the values of the ephemeral keys *ephPrivKey* and *ephPubKey* are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCBVerifyDSA

Verifies authenticity of the digital signature over a message digest (ECDSA).

Syntax

```
IppStatus ippsECCBVerifyDSA(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pSignX, const IppsBigNumState* pSignY,
    IppECResult* pResult, IppsECCBState* pECC);
```

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pSignX`	Pointer to the integer r of the digital signature.
`pSignY`	Pointer to the integer s of the digital signature.

`pResult`	Pointer to the digital signature verification result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function verifies authenticity of the digital signature over a message digest *msg*. The signature consists of two large integers: *r* and *s*.

The scheme used to verify the signature is an elliptic curve analogue of the DSA scheme and assumes that the following cryptosystem key be hitherto set:

regPubKey	Message sender's regular public key.

The *regPubKey* is set by the function `ippsECCBSetKeyPair`.

The result of the digital signature verification can take one of two possible values:

`IppsECSignIsValid`	Digital signature is valid.
`IppsECSignIsInvalid`	Digital signature is not valid.

The call to the `ippsECCBVerifyDSA` function must be preceded by the call to the `ippsECCBSignDSA` function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pMsgDigest`, `pSignX`, `pSignY`, or `ECC` is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by `pMsgDigest` falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base base point *G*.

ECCBSignNR

Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsECCBSignNR(const IppsBigNumState* pMsgDigest, IppsBigNumState*
    pSignX, IppsBigNumState* pSignY, IppsECCBState*   pECC);
```

Parameters

pMsgDigest	Pointer to the message digest *msg*.
pSignX	Pointer to the integer *r* of the digital signature.
pSignY	Pointer to the integer *s* of the digital signature.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function computes two large numbers *r* and *s* which form the digital signature over a message digest *msg*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys are hitherto set up by the message sender:

regPrivKey	Regular private key.
ephPrivKey	Ephemeral private key.
ephPubKey	Ephemeral public key.

The keys can be generated and set up by the functions `ippsECCBGenKeyPair` and `ippsECCBSetKeyPair` with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by `pMsgDigest`, `pSignX`, `pSignY`, or `ECC` is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by `pMsgDigest` falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base point *G*.
`ippStsRangeErr`	Indicates an error condition if one of the parameters pointed by `pSignX` or `pSignY` has a less memory size than the order *n* of the elliptic curve base point *G*.
`ippStsEphemeralKeyErr`	Indicates an error condition if the values of the ephemeral keys *ephPrivKey* and *ephPubKey* are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCBVerifyNR

Computes authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsECCBVerifyNR(const IppsBigNumState* pMsgDigest, const
    IppsBigNumState* pSignX, const IppsBigNumState* pSignY,
    IppECResult* pResult, IppsECCBState* pECC);
```

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pSignX`	Pointer to the integer *r* of the digital signature.
`pSignY`	Pointer to the integer *s* of the digital signature.
`pResult`	Pointer to the digital signature verification result.
`pECC`	Pointer to the context of the elliptic cryptosystem.

Description

The function verifies authenticity of the digital signature over a message digest *msg*. The signature is presented with two large integers *r* and *s*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys be hitherto set up by the message sender:

regPubKey	Message sender's regular private key.

The key can be generated and set up by the function `ippsECCBGenKeyPair`.

The result of the digital signature verification can take one of two possible values:

`IppsECSignIsValid`	The digital signature is valid.
`IppsECSignIsInvalid`	The digital signature is not valid.

The call to the `ippsECCBVerifyNR` function must be preceded by the call to the `ippsECCBSignNR` function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ippsECCBSet` or `ippsECCBSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is NULL.
`ippStsContextMatchErr`	Indicates an error condition if one of the contexts pointed by *`pMsgDigest`*, *`pSignX`*, *`pSignY`*, or *`ECC`* is not valid.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed by *`pMsgDigest`* falls outside the range of [1, 1-*n*] where *n* is the order of the elliptic curve base point *G*.

Functions Removed from Intel® Integrated Performance Primitives 5.0

A

The table below lists functions removed from Intel® Integrated Performance Primitives 5.0 (Intel® IPP 5.0) for cryptography and suggests Intel IPP 5.0 functions to be called instead.

Table A-1 Functions Removed from Intel IPP 5.0

Removed Function	Function for Substitution or Workaround
ippsBigNumBufferSize	ippsBigNumGetSize
ippsBlowfishBufferSize	ippsBlowfishGetSize
ippsDAABlowfishBufferSize	ippsDAABlowfishGetSize
ippsDAADESBufferSize	ippsDAADESGetSize
ippsDAARijndael128BufferSize	ippsDAARijndael128GetSize
ippsDAARijndael192BufferSize	ippsDAARijndael192GetSize
ippsDAARijndael256BufferSize	ippsDAARijndael256GetSize
ippsDAATDESBufferSize	ippsDAATDESGetSize
ippsDAATwofishBufferSize	ippsDAATwofishGetSize
ippsDESBufferSize	ippsDESGetSize
ippsDESDecrypt	ippsDESDecryptECB(...*length*==8...)
ippsDESDecrypt_I	ippsDESDecryptECB(...*length*==8...)
ippsDESEncrypt	ippsDESEncryptECB(...*length*==8...)
ippsDESEncrypt_I	ippsDESEncryptECB(...*length*==8...)
ippsDSABufferSizes	ippsDLPGetSize
ippsDSAInit	ippsDLPInit
ippsDSAKeyCheck	ippsDLPValidateDSA
ippsDSAKeyGen	ippsDLPGenerateDSA

Table A-1 Functions Removed from Intel IPP 5.0 (continued)

Removed Function	Function for Substitution or Workaround
ippsDSAKeyGet	ippsDLPGet
ippsDSAKeySet	ippsDLPSet
ippsDSASign	ippsDLPSignDSA
ippsDSAVerify	ippsDLPVerifyDSA
ippsHMACMD5BufferSize	ippsHMACMD5GetSize
ippsHMACSHA1BufferSize	ippsHMACSHA1GetSize
ippsHMACSHA256BufferSize	ippsHMACSHA256GetSize
ippsHMACSHA384BufferSize	ippsHMACSHA384GetSize
ippsHMACSHA512BufferSize	ippsHMACSHA512GetSize
ippsMD5BufferSize	ippsMD5GetSize
ippsMontBufferSize	ippsMontGetSize
ippsPrimeBufferSize	ippsPrimeGetSize
ippsPRNGAdd	Functionality is not obvious, not needed, removed.
ippsPRNGBufferSize	ippsPRNGGetSize
ippsPRNGGen	ippsPRNGen, ippsPRNGen_BN
ippsPRNGGetRand	ippsPRNGen, ippsPRNGen_BN
ippsPRNGSetPrimeQ	ippsPRNGSetModulus
ippsRijndael128BufferSize	ippsRijndael128GetSize
ippsRijndael192BufferSize	ippsRijndael192GetSize
ippsRijndael256BufferSize	ippsRijndael256GetSize
ippsRSABufferSizes	ippsRSAGetSize
ippsRSAKeyCheck	ippsRSAValidate
ippsRSAKeyGen	ippsRSAGenerate
ippsRSAKeyGet	ippsRSAGetKey
ippsRSAKeySet	ippsRSASetKey
ippsSHA1BufferSize	ippsSHA1GetSize
ippsSHA256BufferSize	ippsSHA256GetSize
ippsSHA384BufferSize	ippsSHA384GetSize
ippsSHA512BufferSize	ippsSHA512GetSize
ippsTDESDecrypt	ippsTDESDecryptECB(…*length*==8...)
ippsTDESDecrypt_I	ippsTDESDecryptECB(…*length*==8...)

Table A-1 Functions Removed from Intel IPP 5.0 (continued)

Removed Function	Function for Substitution or Workaround
ippsTDESEncrypt	ippsTDESEncryptECB(…*length*==8...)
ippsTDESEncrypt_I	ippsTDESEncryptECB(…*length*==8...)
ippsTwofishBufferSize	ippsTwofishGetSize

Subsidiary Source Code Used in Examples

B

The appendix presents subsidiary source code of functions and classes used in Example 5-8 and Example 5-9 given in the Public Key Cryptography Functions chapter.

BigNumber Class

The section presents source code of the `BigNumber` class.

Declarations

Contents of the header file (`bignum.h`) declaring the `BigNumber` class is presented below:

```
#if !defined _BIGNUMBER_H_
#define _BIGNUMBER_H_

#include "ippcp.h"

#include <iostream>
#include <vector>
#include <iterator>
using namespace std;


class BigNumber
{
public:
   BigNumber(Ipp32u value=0);
   BigNumber(Ipp32s value);
   BigNumber(const IppsBigNumState* pBN);
   BigNumber(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);
   BigNumber(const BigNumber& bn);
```

```
BigNumber(const char *s);
virtual ~BigNumber();

// set value
void Set(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);

// conversion to IppsBigNumState
friend IppsBigNumState* BN(const BigNumber& bn) {return bn.m_pBN;}
operator IppsBigNumState* () const { return m_pBN; }

// some useful constatns
static const BigNumber& Zero();
static const BigNumber& One();
static const BigNumber& Two();

// arithmetic operators probably need
BigNumber& operator  = (const BigNumber& bn);
BigNumber& operator += (const BigNumber& bn);
BigNumber& operator -= (const BigNumber& bn);
BigNumber& operator *= (Ipp32u n);
BigNumber& operator *= (const BigNumber& bn);
BigNumber& operator /= (const BigNumber& bn);
BigNumber& operator %= (const BigNumber& bn);
friend BigNumber operator + (const BigNumber& a, const BigNumber& b);
friend BigNumber operator - (const BigNumber& a, const BigNumber& b);
friend BigNumber operator * (const BigNumber& a, const BigNumber& b);
friend BigNumber operator * (const BigNumber& a, Ipp32u);
friend BigNumber operator % (const BigNumber& a, const BigNumber& b);
friend BigNumber operator / (const BigNumber& a, const BigNumber& b);

// modulo arithmetic
BigNumber Modulo(const BigNumber& a) const;
BigNumber ModAdd(const BigNumber& a, const BigNumber& b) const;
BigNumber ModSub(const BigNumber& a, const BigNumber& b) const;
BigNumber ModMul(const BigNumber& a, const BigNumber& b) const;
BigNumber InverseAdd(const BigNumber& a) const;
BigNumber InverseMul(const BigNumber& a) const;

// comparisons
friend bool operator <  (const BigNumber& a, const BigNumber& b);
friend bool operator >  (const BigNumber& a, const BigNumber& b);
friend bool operator == (const BigNumber& a, const BigNumber& b);
friend bool operator != (const BigNumber& a, const BigNumber& b);
friend bool operator <= (const BigNumber& a, const BigNumber& b)
```

```
{return !(a>b);}
   friend bool operator >= (const BigNumber& a, const BigNumber& b)
{return !(a<b);}

   // easy tests
   bool IsOdd() const;
   bool IsEven() const { return !IsOdd(); }

   // size of BigNumber
   int MSB() const;
   int LSB() const;
   int BitSize() const { return MSB()+1; }
   int DwordSize() const { return (BitSize()+31)>>5;}
   friend int Bit(const vector<Ipp32u>& v, int n);

   // conversion and output
   void num2hex( string& s )   const; // convert to hex string
   void num2vec( vector<Ipp32u>& v ) const; // convert to 32-bit word vector

   friend ostream&   operator << (ostream& os,   const BigNumber& a);

protected:
   bool create(const Ipp32u* pData, int length, IppsBigNumSGN
sgn=IppsBigNumPOS);
   int compare(const BigNumber& ) const;

   IppsBigNumState* m_pBN;
};

// convert bit size into 32-bit words
#define BITSIZE_WORD(n) ((((n)+31)>>5))

#endif // _BIGNUMBER_H_
```

Definitions

C++ definitions for the `BigNumber` class methods are given below, the declarations to be included being presented in the preceding Declarations section:

```
#include "bignum.h"

//////////////////////////////////////////////////////////////////////
//
// BigNumber
//
//////////////////////////////////////////////////////////////////////

BigNumber::~BigNumber()
{
   delete [] (Ipp8u*)m_pBN;
}

bool BigNumber::create(const Ipp32u* pData, int length, IppsBigNumSGN
sgn)
{
   int size;
   ippsBigNumGetSize(length, &size);
   m_pBN = (IppsBigNumState*)( new Ipp8u[size] );
   if(!m_pBN)
      return false;
   ippsBigNumInit(length, m_pBN);
   if(pData)
      ippsSet_BN(sgn, length, pData, m_pBN);
   return true;
}

// constructors
//
BigNumber::BigNumber(Ipp32u value)
{
   create(&value, 1, IppsBigNumPOS);
}

BigNumber::BigNumber(Ipp32s value)
{
   Ipp32s avalue = abs(value);
   create((Ipp32u*)&avalue, 1, (value<0)? IppsBigNumNEG : IppsBigNumPOS);
}

BigNumber::BigNumber(const IppsBigNumState* pBN)
{
   int bnLen;
   ippsGetSize_BN(pBN, &bnLen);
```

```
   Ipp32u* bnData = new Ipp32u[bnLen];
   IppsBigNumSGN sgn;
   ippsGet_BN(&sgn, &bnLen, bnData, pBN);
   //
   create(bnData, bnLen, sgn);
   //
   delete bnData;
}

BigNumber::BigNumber(const Ipp32u* pData, int length, IppsBigNumSGN sgn)
{
   create(pData, length, sgn);
}

static
char HexDigitList[] = "0123456789ABCDEF";

BigNumber::BigNumber(const char* s)
{
   bool neg = '-' == s[0];
   if(neg) s++;
   bool hex = ('0'==s[0]) && (('x'==s[1]) || ('X'==s[1]));

   int dataLen;
   Ipp32u base;
   if(hex) {
      s += 2;
      base = 0x10;
      dataLen = (strlen(s) + 7)/8;
   }
   else {
      base = 10;
      dataLen = (strlen(s) + 9)/10;
   }

   create(0, dataLen);
   *(this) = Zero();
   while(*s) {
      char tmp[2] = {s[0],0};
      Ipp32u digit = strcspn(HexDigitList, tmp);
      *this = (*this) * base + BigNumber( digit );
      s++;
   }
```

```
    if(neg)
       (*this) = Zero()- (*this);
}

BigNumber::BigNumber(const BigNumber& bn)
{
   IppsBigNumSGN sgn;
   int length;
   ippsGetSize_BN(bn.m_pBN, &length);
   Ipp32u* pData = new Ipp32u[length];
   ippsGet_BN(&sgn, &length, pData, bn.m_pBN);
   //
   create(pData, length, sgn);
   //
   delete pData;
}


// set value
//
void BigNumber::Set(const Ipp32u* pData, int length, IppsBigNumSGN sgn)
{
   ippsSet_BN(sgn, length, pData, BN(*this));
}


// constants
//
const BigNumber& BigNumber::Zero()
{
    static const BigNumber zero(0);
    return zero;
}

const BigNumber& BigNumber::One()
{
    static const BigNumber one(1);
    return one;
}

const BigNumber& BigNumber::Two()
{
    static const BigNumber two(2);
    return two;
```

```
}


// arithmetic operators
//
BigNumber& BigNumber::operator =(const BigNumber& bn)
{
   if(this != &bn) {    // prevent self copy
      int length;
      ippsGetSize_BN(bn.m_pBN, &length);
      Ipp32u* pData = new Ipp32u[length];
      IppsBigNumSGN sgn;
      ippsGet_BN(&sgn, &length, pData, bn.m_pBN);
      //
      delete (Ipp8u*)m_pBN;
      create(pData, length, sgn);
      //
      delete pData;
   }
   return *this;
}

BigNumber& BigNumber::operator += (const BigNumber& bn)
{
   int aLength;
   ippsGetSize_BN(BN(*this), &aLength);
   int bLength;
   ippsGetSize_BN(BN(bn), &bLength);
   int rLength = IPP_MAX(aLength,bLength) + 1;

   BigNumber result(0, rLength);
   ippsAdd_BN(BN(*this), BN(bn), BN(result));
   *this = result;
   return *this;
}

BigNumber& BigNumber::operator -= (const BigNumber& bn)
{
   int aLength;
   ippsGetSize_BN(BN(*this), &aLength);
   int bLength;
   ippsGetSize_BN(BN(bn), &bLength);
   int rLength = IPP_MAX(aLength,bLength);
```

```
   BigNumber result(0, rLength);
   ippsSub_BN(BN(*this), BN(bn), BN(result));
   *this = result;
   return *this;
}

BigNumber& BigNumber::operator *= (const BigNumber& bn)
{
   int aLength;
   ippsGetSize_BN(BN(*this), &aLength);
   int bLength;
   ippsGetSize_BN(BN(bn), &bLength);
   int rLength = aLength+bLength;

   BigNumber result(0, rLength);
   ippsMul_BN(BN(*this), BN(bn), BN(result));
   *this = result;
   return *this;
}

BigNumber& BigNumber::operator *= (Ipp32u n)
{
   int aLength;
   ippsGetSize_BN(BN(*this), &aLength);
   BigNumber bn(n);

   BigNumber result(0, aLength+1);
   ippsMul_BN(BN(*this), BN(bn), BN(result));
   *this = result;
   return *this;
}

BigNumber& BigNumber::operator %= (const BigNumber& bn)
{
   BigNumber remainder(bn);
   ippsMod_BN(BN(*this), BN(bn), BN(remainder));
   *this = remainder;
   return *this;
}

BigNumber& BigNumber::operator /= (const BigNumber& bn)
{
   BigNumber quotient(*this);
   BigNumber remainder(bn);
```

```
   ippsDiv_BN(BN(*this), BN(bn), BN(quotient), BN(remainder));
   *this = quotient;
   return *this;
}

BigNumber operator + (const BigNumber& a, const BigNumber& b )
{
   BigNumber r(a);
   return r += b;
}

BigNumber operator - (const BigNumber& a, const BigNumber& b )
{
   BigNumber r(a);
   return r -= b;
}

BigNumber operator * (const BigNumber& a, const BigNumber& b )
{
   BigNumber r(a);
   return r *= b;
}

BigNumber operator * (const BigNumber& a, Ipp32u n)
{
   BigNumber r(a);
   return r *= n;
}

BigNumber operator / (const BigNumber& a, const BigNumber& b )
{
   BigNumber q(a);
   return q /= b;
}

BigNumber operator % (const BigNumber& a, const BigNumber& b )
{
   BigNumber r(b);
   ippsMod_BN(BN(a), BN(b), BN(r));
   return r;
}


// modulo arithmetic
```

```
//
BigNumber BigNumber::Modulo(const BigNumber& a)  const
{
   return a % *this;
}

BigNumber BigNumber::InverseAdd(const BigNumber& a) const
{
   BigNumber t = Modulo(a);
   if(t==BigNumber::Zero())
      return t;
   else
      return *this - t;
}

BigNumber BigNumber::InverseMul(const BigNumber& a) const
{
   BigNumber r(*this);
   ippsModInv_BN(BN(a), BN(*this), BN(r));
   return r;
}

BigNumber BigNumber::ModAdd(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a+b);
   return r;
}

BigNumber BigNumber::ModSub(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a + this->InverseAdd(b));
   return r;
}

BigNumber BigNumber::ModMul(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a*b);
   return r;
}


// comparison
//
int BigNumber::compare(const BigNumber &bn) const
```

```
{
   Ipp32u result;
   BigNumber tmp = *this - bn;
   ippsCmpZero_BN(BN(tmp), &result);
   return (result==IS_ZERO)? 0 : (result==GREATER_THAN_ZERO)? 1 : -1;
}

bool operator <  (const BigNumber &a, const BigNumber &b) { return
a.compare(b) < 0; }
bool operator >  (const BigNumber &a, const BigNumber &b) { return
a.compare(b) > 0; }
bool operator == (const BigNumber &a, const BigNumber &b) { return 0 ==
a.compare(b);}
bool operator != (const BigNumber &a, const BigNumber &b) { return 0 !=
a.compare(b);}


// easy tests
//
bool BigNumber::IsOdd() const
{
   vector<Ipp32u> v;
   num2vec(v);
   return v[0]&1;
}

// size of BigNumber
//
int BigNumber::LSB() const
{
   if( *this == BigNumber::Zero() )
      return 0;

   vector<Ipp32u> v;
   num2vec(v);

   int lsb = 0;
   vector<Ipp32u>::iterator i;
   for(i=v.begin(); i!=v.end(); i++) {
      Ipp32u x = *i;
      if(0==x)
         lsb += 32;
      else {
         while(0==(x&1)) {
```

```
                lsb++;
                x >>= 1;
            }
            break;
        }
    }
    return lsb;
}

int BigNumber::MSB() const
{
    if( *this == BigNumber::Zero() )
        return 0;

    vector<Ipp32u> v;
    num2vec(v);

    int msb  = v.size()*32 -1;
    vector<Ipp32u>::reverse_iterator i;
    for(i=v.rbegin(); i!=v.rend(); i++) {
        Ipp32u x = *i;
        if(0==x)
            msb -=32;
        else {
            while(!(x&0x80000000)) {
                msb--;
                x <<= 1;
            }
            break;
        }
    }
    return msb;
}

int Bit(const vector<Ipp32u>& v, int n)
{
    return 0 != ( v[n>>5] & (1<<(n&0x1F)) );
}


// conversions and output
//
void BigNumber::num2vec( vector<Ipp32u>& v ) const
{
```

```
   int length;
   ippsGetSize_BN(BN(*this), &length);
   Ipp32u* pData = new Ipp32u[length];
   IppsBigNumSGN sgn;
   ippsGet_BN(&sgn, &length, pData, BN(*this));
   //
   for(int n=0; n<length; n++)
      v.push_back( pData[n] );
   //
   delete pData;
}


void BigNumber::num2hex( string& s ) const
{
   int length;
   ippsGetSize_BN(BN(*this), &length);
   Ipp32u* pData = new Ipp32u[length];
   IppsBigNumSGN sgn;
   ippsGet_BN(&sgn, &length, pData, BN(*this));

   s.append(1, (sgn==IppsBigNumNEG)? '-' : ' ');
   s.append(1, '0');
   s.append(1, 'x');
   for(int n=length; n>0; n--) {
      Ipp32u x = pData[n-1];
      for(int nd=8; nd>0; nd--) {
         char c = HexDigitList[(x>>(nd-1)*4)&0xF];
         s.append(1, c);
      }
   }
   delete pData;
}

ostream& operator << ( ostream &os, const BigNumber& a) {
   string s;
   a.num2hex(s);
   os << s.c_str();
   return os;
}
```

Functions for Creation of Cryptographic Contexts

The section presents source code for creation of some cryptographic contexts.

Declarations

Contents of the header file (`cpobjs.h`) declaring functions for creation of some cryptographic contexts is presented below:

```
#if !defined _CPOBJS_H_
#define _CPOBJS_H_

//
// create new of some ippCP 'objects'
//
#include "ippcp.h"
#include <stdlib.h>

#define BITS_2_WORDS(n) (((n)+31)>>5)
int Bitsize2Wordsize(int nBits);

Ipp32u* rand32(Ipp32u* pX, int size);

IppsBigNumState* newBN(int len, const Ipp32u* pData=0);
IppsBigNumState* newRandBN(int len);
void deleteBN(IppsBigNumState* pBN);

IppsPRNGState* newPRNG(int seedBitsize=160);
void deletePRNG(IppsPRNGState* pPRNG);

IppsRSAState* newRSA(int lenN, int lenP, IppRSAKeyType type);
void deleteRSA(IppsRSAState* pRSA);

IppsDLPState* newDLP(int lenM, int lenL);
void deleteDLP(IppsDLPState* pDLP);

#endif // _CPOBJS_H_
```

Definitions

C++ definitions of functions creating cryptographic contexts are given below, the declarations to be included being presented in the preceding Declarations section:

```
#include "cpobjs.h"


// convert bitsize into 32-bit wordsize
int Bitsize2Wordsize(int nBits)
{ return (nBits+31)>>5; }


// new BN number
IppsBigNumState* newBN(int len, const Ipp32u* pData)
{
   int size;
   ippsBigNumGetSize(len, &size);
   IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [size] );
   ippsBigNumInit(len, pBN);
   if(pData)
      ippsSet_BN(IppsBigNumPOS, len, pData, pBN);
   return pBN;
}

// desrtoy BN
void deleteBN(IppsBigNumState* pBN)
{ delete [] (Ipp8u*)pBN; }


// set up array of 32-bit items random
Ipp32u* rand32(Ipp32u* pX, int size)
{
   for(int n=0; n<size; n++)
      pX[n] = (rand()<<16) + rand();
   return pX;
}

IppsBigNumState* newRandBN(int len)
{
   Ipp32u* pBuffer = new Ipp32u [len];
   IppsBigNumState* pBN = newBN(len, rand32(pBuffer,len));
   delete [] pBuffer;
   return pBN;
}


//
```

```
// 'external' PRNG
//
IppsPRNGState* newPRNG(int seedBitsize)
{
   int seedSize = Bitsize2Wordsize(seedBitsize);
   Ipp32u* seed = new Ipp32u [seedSize];
   Ipp32u* augm = new Ipp32u [seedSize];

   int size;
   IppsBigNumState* pTmp;
   ippsPRNGGetSize(&size);
   IppsPRNGState* pCtx = (IppsPRNGState*)( new Ipp8u [size] );
   ippsPRNGInit(seedBitsize, pCtx);

   ippsPRNGSetSeed(pTmp=newBN(seedSize,rand32(seed,seedSize)), pCtx);
   delete [] (Ipp8u*)pTmp;
   ippsPRNGSetAugment(pTmp=newBN(seedSize,rand32(augm,seedSize)), pCtx);
   delete [] (Ipp8u*)pTmp;

   delete [] seed;
   delete [] augm;
   return pCtx;
}
void deletePRNG(IppsPRNGState* pPRNG)
{ delete [] (Ipp8u*)pPRNG; }


//
// RSA context
//
IppsRSAState* newRSA(int lenN, int lenP, IppRSAKeyType type)
{
   int size;
   ippsRSAGetSize(lenN,lenP, type, &size);
   IppsRSAState* pCtx = (IppsRSAState*)( new Ipp8u [size] );
   ippsRSAInit(lenN,lenP, type, pCtx);

   return pCtx;
}
void deleteRSA(IppsRSAState* pRSA)
{ delete [] (Ipp8u*)pRSA; }


//
```

```
// DLP context
//
IppsDLPState* newDLP(int lenM, int lenL)
{
   int size;
   ippsDLPGetSize(lenM, lenL, &size);
   IppsDLPState *pCtx= (IppsDLPState *)new Ipp8u[ size ];
   ippsDLPInit(lenM, lenL, pCtx);

   return pCtx;
}
void deleteDLP(IppsDLPState* pDLP)
{ delete [] (Ipp8u*)pDLP; }
```

Bibliography

This bibliography provides a list of publications that might be helpful to you in using cryptography functions of Intel IPP.

[AC] B. Schneier. *Applied Cryptography. Protocols, Algorithms, and Source Code in C. Second Edition*. John Wiley & Sons, Inc., 1996.

[AES] J. Daemon and V. Rijmen. *The Rijndael Block Cipher. AES Proposal.* Available from http://www.nist.gov/aes.

[ANSI] *ANSI X9.62-1998 Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA).* American Bankers Association, 1999.

[BF] B. Schneier. *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)*.
Available from http://www.schneier.com/blowfish.html.

[FIPS PUB 46-3] *Federal Information Processing Standards Publications, FIPS PUB 46-3*. Data Encryption Standard (DES), October 1999. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 113] *Federal Information Processing Standards Publications, FIPS PUB 113*. Computer Data Authentication, May 1985. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 180-2] *Federal Information Processing Standards Publications, FIPS PUB 180-2*. Secure Hash Standard, August 2002. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 186-2] *Federal Information Processing Standards Publications, FIPS PUB 186-2*. Digital Signature Standard (DSS), January 2000. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 198] *Federal Information Processing Standards Publications, FIPS PUB 198*. The Key-Hash Message Authentication Code (HMAC), March 2002. Available from http://csrc.nist.gov/publications/fips.

[IEEE P1363A] *Standard Specifications for Public-Key Cryptography: Additional Techniques.* May, 2000. Working Draft.

[NIST SP 800-38A] *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. NIST Special Publication 800-38A, December 2001. Available from http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[NIST SP 800-38C] *Draft Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality.* NIST Special Publication 800-38C, September 2003. Available from http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf

[PKCS 1.2.1] *RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard*, June 2002. Available from http://www.rsasecurity.com/rsalabs/pkcs.

[PKCS 7] *RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard*, An RSA Laboratories Technical Note Version 1.5 Revised, November 1, 1993.

[RFC 1321] R. Rivest, *"The MD5 Message-Digest Algorithm"*, RFC 1321, MIT and RSA Data Security, Inc, April 1992. Available from http://www.faqs.org/rfc1321.html.

[RFC2401] H.Krawczyk, M. Bellare and R. Canetti, *"HMAC: Keyed-Hashing for Message Authentication"*, RFC 2401, February 1997. Available from http://www.faqs.org/rfcs/rfc2401.html.

[SEC1] *SEC1: Elliptic Curve Cryptography.* Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm.

[SEC2] *SEC2: Recommended Elliptic Curve Domain Parameters.* Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm/.

[TF] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N.Ferguson. *Twofish: A 128-Bit Block Cipher.* Available from http://www.counterpane.com/twofish.html.

[X9.42] *X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.* American National Standards Institute, 2003.

Index

A

B

C

D

E

F

H

I

N

O

P

R

S

T